CLARIS®

HyperCard®

Script Language Guide

T. NELSON
1990

HyperCard
Script Language Guide

©1989-1990 Apple Computer, Inc. All Rights Reserved.

Claris, MacPaint and MacWrite are registered trademarks of Claris Corporation.

HyperCard and HyperTalk are registered trademarks of Apple Computer, Inc., licensed to Claris Corporation.

Apple, Apple logo, ImageWriter, LaserWriter, Mac, Macintosh, MPW, MultiFinder and SANE are registered trademarks of Apple Computer, Inc.

Finder, ResEdit and QuickDraw are trademarks of Apple Computer, Inc.

Addison-Wesley is a registered trademark of Addison-Wesley Publishing Co.

Hayes is a registered trademark of Hayes Microcomputer Products, Inc.

Helvetica and Palatino are registered trademarks of Linotype AG and/or its subsidiaries.

LIMITED WARRANTY AND DISCLAIMERS

A. Limited Warranty on Media. Claris warrants the disks on which the Software is recorded to be free from defects in materials and faulty workmanship under normal use for a period of ninety (90) days from the date of delivery as evidenced by a copy of the sales receipt. Claris will, at its option, replace or refund the purchase price of the diskette at no charge to you, provided you return the faulty diskette with the sales receipt to Claris or an authorized Claris Dealer. Claris shall have no responsibility to replace or refund the purchase price of a diskette damaged by accident, abuse, or misapplication. ANY IMPLIED WARRANTIES ON THE DISKS, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO 90 DAYS FROM THE DATE OF DELIVERY.

B. Disclaimer of Warranty on Software. THE SOFTWARE IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, AND CLARIS EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. CLARIS DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE OR ANY ACCOMPANYING WRITTEN MATERIALS IN TERMS OF THEIR CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS, OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE AND WRITTEN MATERIALS IS ASSUMED BY YOU. CLARIS DOES NOT WARRANT THAT THE SOFTWARE WILL WORK CORRECTLY IN YOUR MULTI-USER OR NETWORK ENVIRONMENT. IF THE SOFTWARE IS DEFECTIVE, YOU, AND NOT CLARIS OR ITS DEALERS, DISTRIBUTORS, AGENTS, OR EMPLOYEES, ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR, OR CORRECTION. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

C. Complete Statement of Warranty. THE LIMITED WARRANTIES PROVIDED IN PARAGRAPHS A AND B ABOVE ARE THE ONLY WARRANTIES OF ANY KIND THAT ARE MADE BY CLARIS ON THIS CLARIS PRODUCT. NO ORAL OR WRITTEN INFORMATION OR ADVICE GIVEN BY CLARIS, ITS DEALERS, DISTRIBUTORS, AGENTS, OR EMPLOYEES SHALL CREATE A WARRANTY OR IN ANY WAY INCREASE THE SCOPE OF THIS WARRANTY, AND YOU MAY NOT RELY ON ANY SUCH INFORMATION OR ADVICE. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS. YOU MAY HAVE OTHER RIGHTS, WHICH VARY FROM STATE TO STATE.

D. Limitation of Liability. IN NO EVENT WILL CLARIS, OR ITS DEVELOPERS, DIRECTORS, OFFICERS, EMPLOYEES, OR AFFILIATES BE LIABLE TO YOU FOR ANY CONSEQUENTIAL, INCIDENTAL, OR INDIRECT DAMAGES (INCLUDING DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, AND THE LIKE) ARISING OUT OF THE USE OF OR INABILITY TO USE THE SOFTWARE OR ACCOMPANYING WRITTEN MATERIALS, EVEN IF CLARIS OR AN AUTHORIZED CLARIS REPRESENTATIVE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

BECAUSE SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU. Claris' liability to you for actual damages for any cause whatsoever, and regardless of the form of the action, will be limited to the greater of $500 or the money paid for the Software that caused the damages.

U.S. Government Restricted Provisions

If this Software is acquired by or on behalf of a unit or agency of the United States Government this provision applies. This Software:
(a) Was developed at private expense, and no part of it was developed with government funds,
(b) Is a trade secret of Claris for all purposes of the Freedom of Information Act,
(c) Is "commercial computer software" subject to limited utilization as provided in the contract between the vendor and the governmental entity, and
(d) In all respects is proprietary data belonging solely to Claris.

For units of the Department of Defense (DOD), this Software is sold only with "Restricted Rights" as that term is defined in the DOD Supplement to the Federal Acquisition Regulations, 52.227-7013 (c) (1) (ii) and:

Use, duplication or disclosure is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at 52.227-7013. Manufacturer:
Claris Corporation
5201 Patrick Henry Drive Box 58168
Santa Clara, CA 95052-8168.

If this Software was acquired under a GSA Schedule, the U.S. Government has agreed to refrain from changing or removing any insignia or lettering from the Software or the accompanying written materials that are provided or from producing copies of manuals or disks (except one copy for backup purposes) and:
(e) Title to and ownership of this Software and documentation and any reproductions thereof shall remain with Claris,
(f) Use of this Software and documentation shall be limited to the facility for which it is acquired, and:
(g) If use of the Software is discontinued to the installation specified in the purchase/ delivery order and the U.S. Government desires to use it at another location, it may do so by giving prior written notice to Claris, specifying the type of computer and new location site. U.S. Governmental personnel using this Software, other than under a DOD contract or GSA Schedule, are hereby on notice that use of this Software is subject to restrictions which are the same as or similar to those specified above.

Contents

Preface About This Guide xix
What's in this book? xx
Notation conventions xxii
Changes since the first edition of this guide xxii
Apple Developer Programs xxiii

Chapter 1 What's New Since 1.2? 1
HyperCard enhancements 2
Variable card sizes 2
Multiple stacks open at the same time 3
Multiple fonts, font sizes, and font styles within fields 4
Group text 4
Navigation palette 5
User-defined palettes 5
Improved printing 5
Script editor enhancements 6
Built-in script debugging tools 6
User-definable menus 7
Extensions to the XCMD interface 7
Enhanced support for MultiFinder 8
Stacks as shared code libraries 8
Enhanced HyperTalk 9

Chapter 2 HyperTalk Basics 15
What is HyperTalk? 16
Objects 16
Buttons and fields 16
Cards, backgrounds, and stacks 18
Messages 18
Scripts 19
Message handlers 19
Function handlers 20
Windows 21
Card windows 21
HyperCard built-in external windows 24
Menus 24
Chapter summary 26

Chapter 3 The Scripting Environment 27
Getting to the script 28
The script editor 31
Manipulating text 33
Searching for text 33
Replacing text 34
Entering comments 35
Formatting scripts 36
Line length and script size 37
Script editor command summary 38
The debugger environment 39
Setting checkpoints 41
HyperTalk debugger windows 42
Message Watcher 42
Variable Watcher 43
Custom message watcher and variable watcher XCMDs 45
Debugger command summary 45
Chapter summary 46

Chapter 4 Handling Messages 47
The HyperCard environment 48
Sending messages 48
System messages 49
Statements as messages 50
Message box messages 50
Messages resulting from commands 51
Receiving messages 51
Message-passing hierarchy 53
Where messages go 53
Messages to buttons and fields 54
The current hierarchy 56
The target 58
The user-defined hierarchy 59
The dynamic path 63
The `go` command and the dynamic path 63
The `send` keyword and the dynamic path 65
Handlers calling handlers 67
Subroutine calls 67
Recursion 68
Using the hierarchy 69
Sharing handlers 69
Intercepting messages 71
Parameter passing 73
Chapter summary 75

Chapter 5 Referring to Objects, Menus, and Windows 77
Names, numbers, and IDs 78
Object names 79
Object ID numbers 80
Object numbers 81
Special ordinals 82
Object numbers and tab order 82
Special object descriptors 82

Identifying a stack 83
Naming a stack 85
Combining object descriptors 86
Referring to menus and menu items 87
Menu and menu item names 87
Menu and menu item numbers 87
Referring to windows 90
Chapter summary 91

Chapter 6 Values 93
Constants 94
Literals 94
Functions 95
Properties 95
Numbers 96
Standard Apple Numerics Environment 96
Precision 97
Number handling 97
Containers 98
Fields 98
Variables 99
Scope of variables 99
Parameter variables 100
The variable `It` 100
The selection 100
The Message box 102
Chapter summary 103

Chapter 7 Expressions 105
Complex expressions 106
Factors 106
HyperTalk operators 107
Operator precedence 111
Operators and expression type 112

Chunk expressions 112
Syntax of chunk expressions 113
Characters 114
Words 114
Items 114
Lines 115
Ranges 116
Chunks and containers 117
Chunks as destinations as well as sources 119
Nonexistent chunks 119
Chapter summary 120

Chapter 8 System Messages 121
Messages and commands 122
Messages sent to a button 122
Messages sent to a field 124
Messages sent to the current card 126
Message order 132

Chapter 9 Control Structures and Keywords 135
Keywords in message handlers 136
On 137
End 137
Exit 138
Pass 138
Return 138
Message handler example 139
Keywords in function handlers 140
Function 141
End 141
Exit 141
Pass 142
Return 142
Function handler example 143

Repeat structure 144
Repeat statements 144
Repeat forever 144
Repeat for 145
Repeat until 145
Repeat while 146
Repeat with 146
Exit repeat statement 147
Next repeat statement 147
End repeat statement 148
If structure 148
Single-statement If structure 148
Multiple-statement If structure 150
Nested If structures 151
Do 152
Global 152
Send 153

Chapter 10 Commands 155
Redefining commands 156
Syntax description notation 156
Add 158
Answer 159
ArrowKey 162
Ask 164
Beep 166
Choose 167
Click 169
Close file 171
Close printing 172
Close window 173
ControlKey 174
Convert 176
Create menu 178

Create stack 179
Debug checkpoint 180
Delete 181
Dial 183
Disable 184
Divide 185
Do 186
DoMenu 187
Drag 189
Edit script 190
Enable 191
EnterKey 192
Export paint 193
Find 194
FunctionKey 198
Get 199
Go 200
Help 201
Hide 202
Import paint 205
Lock screen 206
Mark 207
Multiply 209
Open 210
Open file 212
Open printing 213
Open report printing 214
Palette 216
Picture 218
Play 223
Pop card 224
Print 225
Print card 227
Push 229
Put 230

Read 233
Reset menuBar 235
Reset paint 236
Reset printing 236
ReturnKey 237
Save 237
Select 238
Set 239
Show 240
Show cards 243
Sort 244
Start using 246
Stop using 247
Subtract 247
TabKey 248
Type 249
Unlock screen 250
Unmark 251
Visual 252
Wait 254
Write 255

Chapter 11 Functions 257
Function calls 258
Syntax description notation 259
Abs 260
Annuity 260
Atan 261
Average 262
CharToNum 262
ClickChunk 263
ClickLine 264
ClickLoc 265
ClickText 266

CommandKey 271
Compound 268
Cos 269
Date 270
DiskSpace 271
Exp 272
Exp1 272
Exp2 273
FoundChunk 273
FoundField 274
FoundLine 275
FoundText 275
Length 276
Ln 276
Ln1 277
Log2 277
Max 278
Menu 279
Menus 279
Min 280
Mouse 281
MouseClick 282
MouseH 283
MouseLoc 283
MouseV 284
Number 285
NumToChar 286
Offset 288
OptionKey 289
Param 290
ParamCount 291
Params 292
Random 293
Result 294
Round 296

ScreenRect 296
Seconds 297
SelectedChunk 298
SelectedField 299
SelectedLine 300
SelectedLoc 301
SelectedText 302
ShiftKey 303
Sin 303
Sound 304
Sqrt 305
Stacks 305
Tan 306
Target 307
Ticks 308
Time 309
Tool 310
Trunc 311
Value 312
Windows 313

Chapter 12 Properties 315
Retrieving and setting properties 316
Object properties 316
Stack properties 317
Background properties 318
Card properties 319
Field properties 320
Button properties 322
Rectangle properties 324
Global properties 326
Painting properties 329
Window properties 330
Menu and menu item properties 331

Message Watcher and Variable Watcher properties 332
HyperCard property descriptions 333
AutoHilite 334
AutoTab 336
BlindTyping 337
Bottom 338
BottomRight 339
Brush 340
CantAbort 341
CantDelete 342
CantModify 343
CantPeek 344
Centered 345
CheckMark 346
CmdChar 347
Cursor 348
Debugger 349
DontSearch 350
DontWrap 351
DragSpeed 352
EditBkgnd 353
Enabled 354
Filled 355
FixedLineHeight 356
FreeSize 357
Grid 358
HBarLoc 359
Height 360
HideIdle 361
HideUnused 362
Hilite 363
Icon 364
ID 365
Language 366
Left 367

LineSize 368
Location 369
LockMessages 371
LockRecent 372
LockScreen 373
LockText 374
LongWindowTitles 375
MarkChar 376
Marked 377
MenuMsg 378
MessageWatcher 380
Multiple 381
MultiSpace 382
Name 383
Number 385
NumberFormat 386
Pattern 387
PolySides 388
PowerKeys 389
PrintMargins 390
PrintTextAlign 391
PrintTextFont 392
PrintTextHeight 393
PrintTextSize 394
PrintTextStyle 395
Rect 396
Rectangle 397
ReportTemplates 400
Right 401
Script 402
ScriptEditor 403
ScriptTextFont 404
ScriptTextSize 405
ScriptTextStyle 405
Scroll (windows) 407

Scroll (fields) 408
SharedHilite 410
SharedText 411
ShowLines 412
ShowName 413
ShowPict 414
Size 415
StacksInUse 416
Style 417
TextAlign 418
TextArrows 419
TextFont 420
TextHeight 421
TextSize 422
TextStyle 424
TextStyle (menu items) 426
Top 427
TopLeft 428
TraceDelay 429
UserLevel 430
UserModify 431
VariableWatcher 432
VBarLoc 433
Version 434
Visible 436
WideMargins 438
Width 439

Appendix A External Commands and Functions 441
Definitions, uses, and examples 442
XCMD and XFCN resources 442
Uses for XCMDs and XFCNs 443
Using an XCMD or XFCN 443
Invoking XCMDs and XFCNs 443

Message-passing hierarchy 444
Guidelines for writing XCMDs and XFCNs 446
Attaching an XCMD or XFCN 447
Parameter block data structure 448
Passing parameters to XCMDs and XFCNs 449
ParamCount 449
Params 449
Passing back results to HyperCard 449
ReturnValue 449
PassFlag 448
Callbacks 450
EntryPoint 450
Request 450
Result 451
InArgs 451
OutArgs 451
Callback procedures and functions 451
HyperTalk utilities 452
Memory utilities 454
String utilities 454
String conversions 455
Field utilities 457
Creating and disposing of external windows 462
Window utilities 464
Text editing utilities 469
Script editor utilities 470
Variable watcher support 472
Debugger support 473
External windows 474
Events in external windows 475
Handling Events 476
Closing an external window 481
Special XCmdBlock values 481
Message Watcher 481
Variable Watcher 482

Script editor 482
Debugger 482
XtalkObject 483
Window layer management 484
Flash: an example XCMD 487
Flash listing in MPW Pascal 487
Flash listing in MPW C 489
Flash listing in 68000 assembly language 491

Appendix B Constants 493

Appendix C Enhancing the Execution Speed 495
Change stacks as seldom as possible 496
Use variables, not fields, for operations 497
Refer to a remote card rather than going there 498
Migrate to XCMDs and XFCNs for repetitive tasks 499
Set lockScreen to true to avoid needless redrawing 499
Set lockMessages to true during card-to-card data collection 499
Combine multiple messages 500
Take unnecessary code out of loops 500
Use in-line statements rather than handler calls 501
Do complex calculations once 501
Watch overuse of variable references 501

Appendix D Extended ASCII Table 503

Appendix E Operator Precedence Table 507

Appendix F HyperCard Synonyms 509

Appendix G HyperCard Limits 511

Appendix H HyperCard Syntax Summary 515
Syntax description notation 516

Appendix I HyperTalk Vocabulary 531

Glossary 561

Preface **About This Guide**

This book provides detailed information about the HyperTalk® scripting language in the HyperCard® application. Even a little knowledge of HyperTalk enables you to customize buttons and other parts of HyperCard stacks for your own purposes, and you can use HyperTalk to make the stacks that you create act the way you want.

While you're using HyperCard, you can find information about HyperTalk in the HyperCard Help stack and the HyperTalk Reference stack. These stacks make use of some of the best features of HyperCard, such as multiple windows, computer-supported cross-referencing, and fast text searching.

Some of the concepts in this book, such as message handling and objects, may be new to you. Use this guide as it suits your own style of learning: you might be the kind of person who understands best by thoroughly studying the explanations, or you might be the kind who learns by skimming the material and then using HyperTalk—writing scripts or copying the examples and trying them out.

The Claris® HyperCard Development Kit contains reference material for beginning scriptors in the *HyperCard Reference Manual* and the *HyperTalk Beginner's Guide*. Contact your Claris authorized dealer to purchase the Claris HyperCard Development Kit.

What's in this book?

Here's a brief description of the contents of this book:

Chapter 1, "What's New Since 1.2?," discusses the differences between earlier versions of HyperCard and HyperCard 2.0. Those of you who are already familiar with HyperTalk as described in the original edition of this book can use this chapter as a guide to new information in this edition. If you're new to HyperTalk, however, and haven't used the original edition, you'll probably want to skip Chapter 1 initially, and come back to it later.

Chapter 2, "HyperTalk Basics," introduces the basic concepts of HyperTalk, showing how it is used in the HyperCard environment.

Chapter 3, "The Scripting Environment," explains how to create and modify scripts in HyperCard objects.

Chapter 4, "Handling Messages," describes how HyperTalk works, how it carries out actions, and how it responds to events in the HyperCard environment.

Chapter 5, "Referring to Objects, Menus, and Windows," explains how to refer to objects—the parts of HyperCard that contain HyperTalk scripts and that respond to and initiate actions.

Chapter 6, "Values," explains the elements within HyperTalk that contain values.

Chapter 7, "Expressions," describes the HyperTalk operators and explains how HyperTalk evaluates expressions—the descriptions of how to get a value.

Chapter 8, "System Messages," describes the messages that HyperCard generates in response to events (such as mouse clicks) that happen in its environment.

Chapter 9, "Control Structures and Keywords," describes the handlers within which you write all HyperTalk scripts to enable objects to respond to messages and function calls. It also describes the control structures of HyperTalk that let you specify how and when sections of scripts execute, and it describes the keywords that you use in control structures.

Chapter 10, "Commands," describes each of the HyperTalk built-in commands—the action statements that make HyperCard do things.

Chapter 11, "Functions," describes the HyperTalk built-in functions—named values that reflect conditions in the HyperCard environment.

Chapter 12, "Properties," describes the properties of HyperCard objects—characteristics that determine how objects look and act.

Appendix A, "External Commands and Functions," contains general information about XCMDs and XFCNs, extensions to HyperTalk that can be written by expert programmers to increase the power of HyperCard.

Appendix B, "Constants," describes the HyperTalk built-in constants—named values that don't change.

Appendix C, "Enhancing the Execution Speed of HyperCard," provides some helpful hints for scripters who want to increase the efficiency of HyperCard.

Appendix D, "Extended ASCII Table," lists the decimal values of the standard Macintosh character set used by HyperCard.

Appendix E, "Operator Precedence Table," summarizes the order in which HyperTalk performs operations when it evaluates expressions.

Appendix F, "HyperCard Synonyms," lists the abbreviations and alternate spellings for HyperTalk terms.

Appendix G, "HyperCard Limits," lists various minimum and maximum sizes and numbers of elements defined in HyperCard.

Appendix H, "HyperTalk Syntax Summary," shows the syntax of the HyperTalk command and function parameters in abbreviated form. It also lists and defines the metasymbols used in syntax of the commands, properties, and functions.

Appendix I, "HyperTalk Vocabulary," lists alphabetically each of the primary HyperTalk terms that HyperCard understands, names the category it's in, and provides a brief description of its meaning.

You will also find a glossary of terms commonly associated with the HyperCard environment and an index to help you quickly find specific information contained in this guide.

Notation conventions

Before you read this guide, you should know about a few typographic conventions. Words or phrases in a monospaced font `like this` are HyperTalk language elements or are to be typed exactly as shown. New terms are shown in **boldface** type when they are first introduced and defined. The glossary contains definitions of these terms and other related technical terms.

In descriptions of HyperTalk syntax for commands and other language elements, words in *italic* type describe general elements, not specific names—you must substitute the actual instances. (These elements are called *metasymbols* in this book.) Square brackets ([ ]) enclose optional elements which may be included if you need them. (Don't type the square brackets.) In some cases, optional elements change what the message does; in other cases they are helper words that have no effect except to make the message more readable. Syntax descriptions for some language elements have a particular format, which is explained at the beginning of the chapter about that language element.

It doesn't matter whether you use uppercase or lowercase letters in commands or variable names; message names that are formed from two words are shown in small letters with a capital in the middle (`likeThis`) merely to make them more readable.

Changes since the first edition of this guide

This edition of the HyperCard Script Language Guide is different from the first edition in several ways. Of course, it has new information in it that reflects the new features of HyperCard. In addition, the page format and design are different, and to make finding information easier, a few chapters have been divided into smaller chapters and others have been reorganized.

Chapter 1, "HyperTalk Basics," is now two chapters: Chapter 2, "HyperTalk Basics," which describes HyperTalk, and Chapter 3, "The Scripting Environment," dedicated to the script editor. Chapter 4, "Values," is now Chapter 6, "Values," and Chapter 7, "Expressions." Chapter 5, "Keywords," is now Chapter 9, "Control Structures and Keywords."

Chapter 12, "Properties" now has one main alphabetical list of properties, rather than having the properties grouped by the object or environment to which they can apply. For each property, the objects or environments to which it can apply are listed on the first line after the heading. Also, at the beginning of the chapter, there are tables that list properties by object or environment (button properties, field properties, painting properties, and so forth).

In Chapters 10, 11, and 12, the names of the commands, functions, and properties that are new or enhanced since the original edition of this book are followed with a symbol like the one at the beginning of this paragraph to make locating new information easier.

Apple Developer Programs

Apple offers two programs: the Partners Program, for developers who intend to resell Apple-compatible products, and the Associates Program, for developers who don't intend to resell products and for other people involved in the development of Apple-compatible products.

For more information about the Apple developer support programs, contact Apple Developer Programs at the following address.

Apple Developer Programs
Apple Computer, Inc.
20525 Mariani Avenue, M/S 75-2C
Cupertino, CA 95014

Chapter 1 **What's New Since 1.2?**

This chapter is most useful for those who are already familiar with HyperCard because it describes the enhancements made to HyperCard since HyperCard 1.2.5. If you haven't done any scripting with HyperCard before, you should start with Chapter 2, "HyperTalk Basics," and work your way through the rest of the book. If you are already familiar with HyperCard, you can use this chapter as a guide to locations in the book that cover the new features of HyperTalk in more detail.

HyperCard 2.0 is a major revision of earlier versions of HyperCard. It provides more power and flexibility across the entire range of Macintosh computers, starting with the Macintosh Plus. It incorporates features that improve the use of any HyperCard application on the smaller screen of the Macintosh Plus and Macintosh SE and on larger screens commonly used on the Macintosh II and other modular Macintosh computers.

Many of the new features of HyperCard can be seen in the user interface. For a more detailed description of the user interface, see the *HyperCard Reference.*

- HyperCard version 2.0 and later requires system software version 6.0.5 or later. Stacks created with earlier versions of HyperCard are opened as read-only in HyperCard version 2.0 or later. The earlier-version stacks are write-protected until converted to the 2.0 format by choosing the Convert Stack command from the File menu.

HyperCard enhancements

HyperCard 2.0 provides several major feature enhancements. Most of the features are controllable from the HyperTalk script language and can be easily used to improve the user interface of any HyperCard application. The other features are transparent, but increase the execution speed of HyperCard. The following list summarizes the new features of HyperCard:

- variable card sizes
- multiple stack windows open at the same time
- multiple fonts, font styles, and font sizes within fields
- group text—a hypertext-like feature
- navigation palette
- support for user-defined palettes
- improved printing
- script editor enhancements
- built-in script debugging tools
- user-definable menus
- extensions to the XCMD interface
- enhanced support for MultiFinder
- stacks as shared code libraries
- enhanced HyperTalk

Variable card sizes

The cards in a stack are viewed through a resizable Macintosh-style window. HyperCard 2.0 supports card sizes from 64 x 64 pixels to 1280 x 1280 pixels (approximately 1 inch square to nearly 18 inches square). Card size is set at the time a stack is created. Only one card size is permitted per stack.

You can change a stack's card size anytime from the Stack Info dialog box. However, all of the graphic elements, buttons, and fields that were created within the original card size boundaries remain unchanged. For example, graphics on a smaller card do not expand to fill the new space created when you make a stack's card size larger, and if you choose to make a stack's card size smaller, some graphics and other elements may be completely out of view. The graphics are retained until the stack is compacted.

A stack's card size can also be changed when a stack is compacted or copied. If you reduce the size of a card and then compact a stack, all of the bitmapped graphics (paint text or pictures) on the card or background that are outside the new smaller card size boundaries are lost. The buttons and fields outside the visible boundaries still remain.

The size of the cards in a stack may be limited by the available memory. For example, card sizes greater than 640 x 480 require more memory than the standard Macintosh Plus has—up to 4 MB for authoring at the largest size.

Multiple stack windows open at the same time

You can have multiple stacks open on the screen at the same time. You can open additional stacks from a HyperTalk script or the Message box with the enhanced `go` command by using the form `go stack` *stackName* `in a new window`. The number of stack windows that can be shown on the screen is limited by available RAM up to an absolute limit of 18 stack windows and external windows. The `go` command is described in Chapter 10, "Commands."

The maximum size of a stack's window is the size set for the cards in that stack. Window sizes are constrained if the Macintosh doesn't have enough memory to show the full card. For example, on a 1 MB Macintosh II, the largest window that can be shown is around 640 x 480, regardless of the size of the monitor or the cards in the open stack.

Multiple fonts, font sizes, and font styles within fields

Multiple fonts, font sizes, and font styles within a single field are supported. Font and Style menus have been added to the HyperCard menu bar for changing text in fields. The font, font size, and style of text in fields can be controlled from a HyperTalk script with the enhanced `textFont`, `textSize`, and `textStyle` properties. Text can be cut from a field and pasted into other fields or the scrapbook with all of the style information preserved. Whole fields can be copied with Command-C and then pasted onto any card or background with all of the text and text properties intact by pressing Command-Shift-V when you paste the field.

Fields have several other new properties, including `fixedLineHeight`, `sharedText`, `dontSearch`, and `dontWrap`. Each of the new field properties can be set from the Field Info dialog box, as well as from HyperTalk scripts. For more information about the field properties, see the descriptions in Chapter 12, "Properties."

Group text

Any character, group of characters, lines, or paragraphs can be selected and made into group text through the Style menu or from a script. You use the group style to group words or characters together so they may be used more intelligently by HyperTalk.

A set of HyperTalk functions and the new group style provide a hypertext functionality to HyperCard. The new HyperTalk functions allow you to create handlers that use the existing `mouseUp` and `mouseDown` mechanism to examine and react to group text that has been clicked within locked fields.

For more information about the functions that support group text, see `clickChunk`, `clickField`, `clickLine`, and `clickText` in Chapter 11, "Functions." For more information about the `group` style, see the `textStyle` property in Chapter 12, "Properties," and the `show` and `hide` commands in Chapter 10, "Commands."

Navigation palette

A navigation palette has been added to HyperCard. It provides the first ten functions in the HyperCard Go menu. This palette can eliminate the need for navigation arrow buttons in many stacks. The navigator palette is invoked from the Message box or a script with the HyperTalk command, `palette navigator`. The navigator is implemented through new palette XCMD support in HyperCard.

User-defined palettes

Tools allow you to create your own HyperCard palettes by drawing them with the Paint tools, then creating HyperCard buttons with scripts to send commands from the finished palette to HyperCard.

For more information about creating palettes, see the `palette` command in Chapter 10, "Commands".

Improved printing

HyperCard provides the following new printing features:

- You can print large cards by shrinking to fit or by tiling across several pieces of paper.
- You can print a report from discontinuous groups of cards based on a chosen criteria. For example, you can print all of the cards that contain a specific word, background, or button.
- You can print any field either through a Print Field dialog box or through a HyperTalk script.
- The printing code provides for a greater range of printing devices. Previous versions of HyperCard were optimized for the LaserWriter and ImageWriter.
- You can customize the way a print report looks and save the custom report form as a template.

Summaries of the new HyperTalk properties and commands related to printing are shown in Table 1-1 and Table 1-3.

Script editor enhancements

The enhanced HyperCard script editor is an XCMD that can manage multiple script windows. The new script editor supports having several different scripts open simultaneously and allows cutting and pasting between them.

Features supported by the script editor include undo for cut, copy, paste, clear, and typing operations; horizontal scrolling; search and replace operations; triple-clicking to select a line; setting debugger checkpoints; and commenting and uncommenting of selected lines. See Chapter 3, "The Scripting Environment," for more information about the enhanced script editor.

Built-in script debugging tools

The HyperCard script editor is integrated with a debugger that allows you to trace script execution and set checkpoints. You set checkpoints in HyperTalk handlers with the command `debug checkpoint`. When HyperCard encounters this message, it pauses execution of the handler and opens the script editor window, highlighting the command where the checkpoint is set. Once a checkpoint is invoked, and you are in the debugger, you can step through the remaining lines of the script by pressing Command-S, the editor's Step command. In combination, checkpoints and stepping can be used to examine, diagnose, and modify the behavior of HyperTalk handlers.

You can use Command-D to set or clear temporary checkpoints on any line while editing a script; you can also use Command-D while using the debugger. Temporary checkpoints are never saved to disk but remain in place until HyperCard quits. When HyperTalk encounters a temporary checkpoint, it stops, invokes the debugger, and displays the checkpoint in the script editor window before executing that line of HyperTalk. For more information about the debugging tools, see Chapter 3, "The Scripting Environment."

User-definable menus

User-definable menus are a new feature that gives you control of all HyperCard menus and the ability to create custom menus. Through HyperTalk, you can create new menus with the `create` command and add menu items to menus in the HyperCard menu bar with the enhanced `put` command. Menu items may be enabled, disabled, and deleted with the `enabled` and `disabled` menu properties, `enable` and `disable` commands, and the `delete` command (with the exceptions noted in the next paragraph). Command-key keyboard equivalents and check marks can be created or reassigned to menu items with the `cmdChar` and `markChar` menu properties. Specific HyperTalk messages can be associated with any menu item with the `menuMsg` menu property. The text font and text style of menu items can be set with the `textFont` and `textStyle` menu properties.

The Font, Tools, and Patterns menus are exceptions; they cannot be altered by means of the menu commands and properties, except to be deleted entirely from the menu bar. For more information about user-defined menus, see Chapter 2, "HyperCard Basics," Chapter 5, "Referring to Objects, Menus, and Windows," Chapter 10, "Commands," and Chapter 12, "Properties."

Extensions to the XCMD interface

The extended XCMD interface allows XCMDs to run in parallel with HyperCard. XCMDs can now continue to perform work even though the user may have moved on to other work or even another stack. The new interface also allows XCMDs to run longer in the HyperCard environment than the HyperTalk scripts that called them.

In HyperCard 2.0, existing XCMDs created for earlier versions of HyperCard function as they always have. In addition, HyperCard 2.0 adds support for XCMDs that manage their own windows and use the Sound Manager.

A window that is managed by an XCMD is called an **external window.** Examples of what you can do in external windows include display of color and gray-scale PICT files, support for custom palettes, and creation of custom HyperTalk script editors and debugging tools. HyperCard passes events pertaining to external windows to the XCMDs that created them, thereby eliminating the need for XCMDs to patch into the event loop or to include their own event loops.

The enhancements to the XCMD interface allow you to create very sophisticated add-ons. For more information about the new callbacks for external windows, see Appendix A, "External Commands and Functions."

Enhanced support for MultiFinder

Compacting and sorting of stacks can now be done in the background. Handlers execute in the background. If you switch applications while in the MultiFinder environment the HyperCard handler continues until the operation is complete. While HyperCard is running in the background, it provides information to the Notification Manager so that MultiFinder can let you know if HyperCard requires attention. When HyperCard requires attention the Apple menu icon flashes and a diamond is placed next to HyperCard in the Apple menu.

Stacks as shared code libraries

Stacks can now be used as shared code libraries through a user-defined message-passing hierarchy. HyperTalk messages may pass first through a button or field. All messages are passed through the current card, the current background, the current stack, the Home stack, and finally reach HyperCard itself. Because of this inheritance system, handlers and XCMDs in the Home stack can be called by any script in any other stack. HyperCard 2.0 extends this inheritance scheme beyond the Home stack by allowing you to use the scripts and resources in any other HyperCard stack as shared code libraries. This is accomplished by means of a `start using` statement in HyperTalk. For example:

```
on openStack
  start using stack "HD80:myStack"
end openStack
```

Once this `start using` statement is executed, the handlers in the script of the stack `myStack` and the XCMDs and other resources in its resource fork are available to the current stack. Effectively, `myStack` is inserted in the message-passing hierarchy between the current stack and the Home stack.

Similarly, a stack can be removed from the message-passing hierarchy with a `stop using` statement:

```
on closeStack
  stop using stack "HD80:myStack"
end closeStack
```

Up to ten stacks may be inserted in the message-passing hierarchy by means of the `start using` command.

Enhanced HyperTalk

The HyperTalk interpreter has been replaced with a compiler to increase execution speed. HyperTalk compiles handlers on their first execution and stores the complied code in RAM. As long as the memory space occupied by the compiled handler code is not needed by HyperCard for another purpose, the handler executes from RAM the next time it is called.

The HyperTalk vocabulary has been enlarged to accommodate the new features within HyperCard. For those of you who want to jump right in and look at the new commands, properties, functions, and messages in the enhanced HyperTalk vocabulary, check the tables provided here. See the appropriate chapter for syntax, definitions, and examples.

■ **Table 1-1** Enhanced HyperTalk properties

Property	Definition
`cantAbort`	Stacks; determines whether Command-period can be used to stop execution of a script
`cantPeek`	Stacks; determines whether the user can peek at button or field outlines
`cmdChar`	Menu items; assigns a keyboard equivalent to a menu item
`debugger`	Global; specifies debugger to use
`dontSearch`	Fields; excludes a field from a search with `find` command
`enabled`	Menu items; specifies whether menu item is active or inactive
`fixedLineHeight`	Fields; specifies whether line height is a fixed value or variable

[continued]

■ **Table 1-1** Enhanced HyperTalk properties [continued]

Property	Definition
`hBarLoc`	Variable Watcher window; set the horizontal bar location of the Variable Watcher window
`hideIdle`	Message Watcher window; determines whether the "Hide idle" check box is checked (checked state stops idle message from appearing in the Message Watcher window)
`hideUnused`	Message Watcher window; determines whether the "Hide unused" check box is checked (when checked, only messages intercepted by a handler are displayed)
`longWindowTitles`	Global; determines whether the full pathname or stack name only is shown in the stack title bar
`markChar`	Menus items; specifies the check mark character
`marked`	Cards; determines whether a card is marked; used for complex searching criteria
`menuMessage`	Menu items; assigns a message to a menu item
`messageWatcher`	Global; specifies Message Watcher to use
`name`	Menu items; changes name of a menu item
`printTextFont`	Specifies the text font to use in the header of a print report and when printing variables
`printTextStyle`	Specifies the text style to use in the header of a print report and when printing variables
`printTextSize`	Specifies the text size to use in the header of a print report and when printing variables
`printTextAlign`	Specifies the text alignment to use in the header of a print report and when printing variables
`printTextHeight`	Specifies the line height to use in the header of a print report and when printing variables
`printMargins`	Specifies the margins to use when printing reports and expressions
`rect`	Variable Watcher window; determines size of window

[continued]

■ **Table 1-1** Enhanced HyperTalk properties [continued]

Property	Definition
`rectangle`	Card windows; determines the size of a specified window
`reportTemplates`	Stacks; a read-only property that returns the names of the report templates saved with the specified stack.
`scriptEditor`	Global; specifies the name of script editor to use
`scriptTextFont`	Global; specifies the text font to use in the script editor
`scriptTextSize`	Global; specifies text size to use in the script editor
`scriptTextStyle`	Global; specifies text style to use in the script editor
`scroll`	Cards; point that specifies horizontal and vertical scroll of the card picture
`sharedHilite`	Background buttons; determines whether button has same state on each card of that background
`sharedText`	Background fields; determines whether field text is shared for all cards of that background
`stacksInUse`	Global; determines which stacks are in message-passing hierarchy
`textFont`	Buttons, fields, menus, painting environment; specifies font of text
`textSize`	Buttons, fields, painting environment; specifies size of text
`textStyle`	Buttons, fields, menus, painting environment; specifies style of text
`variableWatcher`	Global; specifies Variable Watcher to use
`vBarLoc`	Variable Watcher window; determines location of the vertical bar in the Variable Watcher window

■ **Table 1-2** Enhanced HyperTalk functions

Function	Definition
`clickChunk`	Returns chunk expression of word or range last clicked
`clickLine`	Returns line expression of last line clicked

[continued]

■ **Table 1-2** Enhanced HyperTalk functions [continued]

Function	Definition
`clickText`	Returns word or range of group text last clicked
`menus`	Returns the names of all menus in the menu bar
`number of marked cards`	Returns count for whole stack
`number of menus`	Returns the count of all menus in the menu bar
`number of windows`	Returns the count of all windows available
`selectedLoc`	Returns the point at which the current text selection begins
`windows`	Returns the names of all windows available

■ **Table 1-3** Enhanced HyperTalk commands

Command
`answer file [`*promptText*`] [of type` *fileType*`]`
`ask password` *question* `[with` *defaultAnswer*`]`
`ask file [`*promptText*`] [with [default]` *fileName*`]`
`create menu` *menuName*
`create stack` *stackName* `[with` *background*`] [in a new window]`
`debug checkpoint`
`delete menu` *menuName*
`delete menuItem` *itemName* `of menu` *menuName*
`disable menuItem` *itemName* `of menu` *menuName*
`enable menuItem` *itemName* `of menu` *menuName*
`go [to]` *card* `[of` *background*`] of [stack]` *stackName*`] [in a new window]` ¬ `[without dialog]`
`hide groups`
`hide titleBar`
`mark` *card* `[cards where` *expression* `| all cards]`
`open report printing [with template` *templateName*`] [with dialog]`
`palette` *paletteName*`[,` *point*`]`
`picture [`*fileName*`,` *fileType*`,` *windowStyle*`,` *visible*`,` *depth*`]`

[continued]

■ **Table 1-3** Enhanced HyperTalk commands [continued]

Command
`print` *card* `[from` *point1* `to` *point2*`]`
`print` *expression*
`print` *field*
`print marked cards`
`put` *itemName preposition* `[`*menuItem* `of]` *menu* `[with menuMsg` *expression*`]`
`reset menubar`
`reset printing`
`save stack` *stackName* `as` *fileName*
`show groups`
`show titlebar`
`show marked cards`
`sort marked cards`
`start using` *stack*
`stop using` *stack*
`unmark` *card* `[where` *expression*`]`
`unmark cards where` *expression*
`unmark all cards`

■ **Table 1-4** Enhanced HyperTalk messages

Message	Definition
`commandKeyDown`	Sent when a key is pressed in combination with the Command key
`doMenu` *itemName, menuName*	Now can specify menu name
`closePalette`	Sent when a palette opened with the `palette` command is closed
`closePicture`	Sent when a window created with the `picture` command is closed

[continued]

■ **Table 1-4** Enhanced HyperTalk messages [continued]

Message	Definition
`keyDown`	Sent when a key is pressed
`mouseDownInPicture` *name, loc*	Sent when the mouse is down in a window created with the `picture` command
`mouseUpInPicture` *name, loc*	Sent when the mouse is up after being down in a window created with the `picture` command
`moveWindow`	Sent when you change the location of a card window
`openPalette`	Sent when a palette is opened with the `palette` command
`openPicture`	Sent when a window is created and opened with the `picture` command
`sizeWindow`	Sent when you change the `rect` property of a card window
`suspendStack`	Sent when you switch to a another stack
`resumeStack`	Sent when you return to an already open stack

■ **Table 1-5** Enhanced HyperTalk visual effects

Effect
`shrink to [top|center|bottom] [`*speed*`]` `stretch from [top|center|bottom] [`*speed*`]`

■ **Table 1-6** HyperTalk global variables

Global	Definition
`scriptWindowRects`	Contains the rectangle parameters to use for script windows

Chapter 2 **HyperTalk Basics**

This chapter explains how HyperTalk works in the HyperCard system and describes some of the HyperTalk characteristics.

Most concepts are discussed only briefly in this chapter, with more detailed discussion left for later chapters.

What is HyperTalk?

HyperTalk is the scripting language of the HyperCard environment. It allows you to perform actions on HyperCard objects: buttons, fields, cards, backgrounds, and stacks, as well as other elements of HyperCard, such as menus and windows.

You use HyperTalk to send messages to and from HyperCard objects. You generate a message by (among other means) clicking the mouse, opening a card, typing a statement into the Message box, or choosing a menu item. How a given object responds to a particular message depends on the object's script, or in the case of menu items, the menu message for that menu item. All HyperCard scripts are written in HyperTalk.

Objects

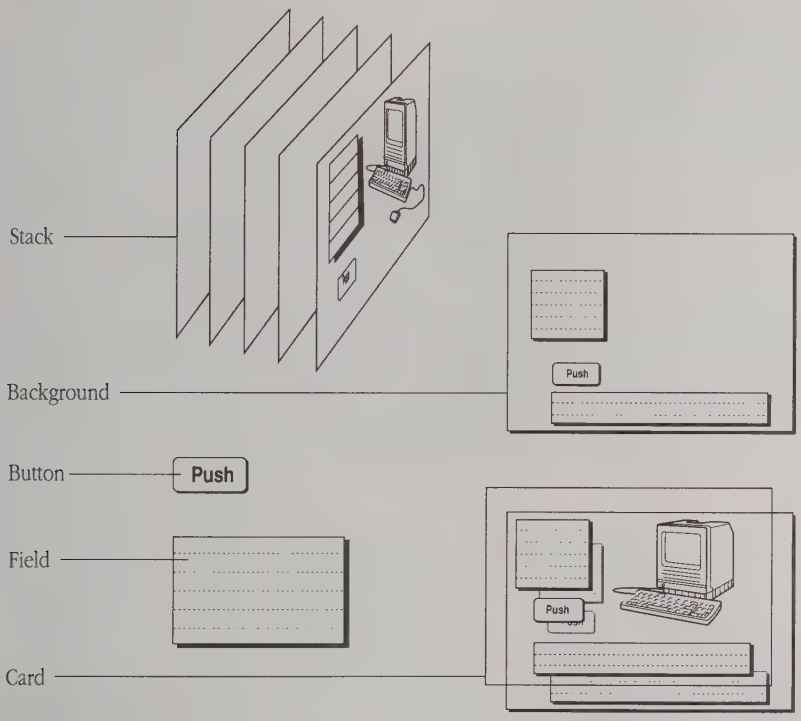
There are five kinds of **objects** in HyperCard: buttons, fields, cards, backgrounds, and stacks. (See Figure 2-1.)

Buttons and fields

Buttons are action objects or "hot spots" on the screen. For example, clicking a button with the Browse tool can take you to the next card in a stack.

Fields contain editable text. The Browse tool pointing hand changes to an I-beam when it's over an unlocked field. (The card or background might also contain Paint text characters. Such characters are not editable once they are placed; they become part of the picture on the card or background.)

■ **Figure 2-1** HyperCard objects



Cards, backgrounds, and stacks

The basic unit of information is the **card:** when you look at the screen of a Macintosh computer running HyperCard, what you see foremost is a card. Cards are viewed through card windows. Cards can be larger than the card window and can be scrolled in the card window with the Scroll command in the Go menu. The **background** is where you place elements that you want a group of cards to have in common. Each card has one background. The card overlays the background; both are the same size. What you see in the card window belongs to the card or to the background. Both the card and background can contain buttons, fields, and pictures. Cards are grouped in **stacks;** each stack is a Macintosh file. Each stack can have multiple backgrounds.

The card that is currently displayed, the background associated with it, and the stack they are in are termed the current card, background, and stack. The concept of being current doesn't apply to buttons or fields. Chapter 5 contains details about referring to objects.

Messages

HyperCard objects interact with each other, with the user, with HyperCard, and with the Macintosh environment by sending messages. Some messages are descriptions of things that happen in the environment, such as that the mouse has been clicked or a card opened: these are **system messages.** They are like news flashes announced to the community of objects. For example, if you press the mouse button down, HyperCard sends the message `mouseDown`; when you let the mouse button up, HyperCard sends the message `mouseUp`. Chapter 8 contains more information about system messages.

Messages are sent to various objects in a particular order. For example, system messages generated by the mouse go first to the topmost button or field (if any) under the pointer on the screen. Next those messages go to the card, then to the background, then the stack, then the Home stack, and finally to HyperCard itself. (You'll find a detailed discussion of this hierarchical sequence in Chapter 4.)

HyperTalk **commands** are also messages—orders to do some particular thing, like add two numbers or go to another card. A command, whether executed in a script or typed into the Message box, is sent as a message.

Scripts

Every HyperCard object has a script (although the script can be completely empty). A **script** is a collection of any number of handlers. A **handler** is a collection of HyperTalk statements; each statement ends with a return character. Any part of a statement following the HyperTalk double-hyphen comment character (`--`) is ignored by HyperCard.

A handler is invoked when a particular message is received by the object whose script contains the handler. A simple handler looks like this:

```
on mouseUp
  go to next card
end mouseUp
```

The first line of a handler always begins with one of two words—either `on` or `function`. The last statement of a handler always begins with the keyword `end`. All HyperTalk statements always appear within handlers in a script.

You must place handlers in the scripts of objects that will receive the messages you want the handlers to respond to. The message-passing hierarchy, which determines where messages are sent, is described in Chapter 4.

Message handlers

A handler that begins with `on` is called a **message handler.** The example in the previous section is a message handler. This particular message handler is in the script of a button; it handles the message `mouseUp`, and goes to the next card.

The message to which a handler responds begins with the word following the word `on`. In this case, the message is `mouseUp`. When you release the mouse button while the Browse tool is inside a button's rectangle on the screen, HyperCard sends the message `mouseUp` to the button. HyperCard looks in the button's script for a handler matching the message `mouseUp`. If it finds a match, it executes the HyperTalk statements between `on mouseUp` and `end mouseUp`—in this case, `go to next card`.

Function handlers

In addition to message handlers, scripts can contain user-defined **function handlers.** Function handlers begin with the word `function` instead of the word `on`; the name of the function they handle is the second word. A function handler looks like this:

```
function day
   return first item of the long date
end day
```

This function handler responds to a HyperTalk statement containing the function's name followed by parentheses—a **function call.** Here's an example:

```
put day() into message box
```

The function call is `day()`—the rest of the line and the function call together form a statement. When the function call is made, HyperCard looks for the matching function handler. If it finds one, it executes the lines between `function day` and `end day`. The value derived from the expression `first item of the long date` is returned to the `put` statement in place of `day()`. In the example, the value returned by the function (`Friday`, for example) is put into the Message box.

Function calls use the same message-passing hierarchy as messages; it's described in Chapter 4. Message and function handler structures are described in detail in Chapter 9.

Windows

Windows are another HyperCard element that you can control with the HyperTalk language. Windows share many properties with objects, but are not HyperCard objects because they do not have scripts or respond to messages. (Scripts and messages are described later in this chapter; see the section "Properties" in Chapter 6, "Values," for an introduction to HyperCard properties.)

HyperCard windows include card windows, the FatBits window, the Tools palette, the Patterns palette, the Scroll window, the Message box window, and external windows.

The built-in external windows of HyperCard include: the script editor, the Message Watcher, the Variable Watcher, and the navigator palette. User-defined external windows include windows created with the `picture` command and windows created with external commands. (The `picture` command is described in Chapter 10, "Commands," and external commands, which control external windows, are described in Appendix A, "External Commands and Functions."

Card windows

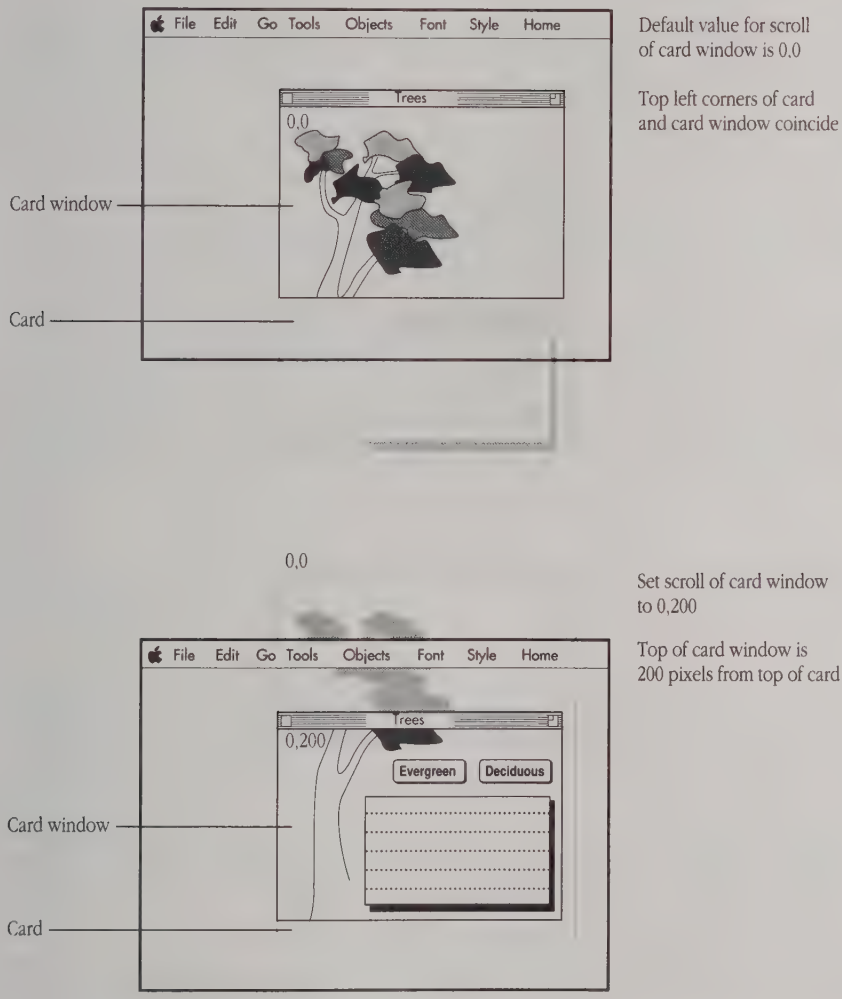
You view the cards in a HyperCard stack through a card window. Card windows have a title bar, a size box, and if there is more than one stack open at the same time, a close box. A stack's card window can be the same size or smaller than the current card size. You can reposition card windows by dragging them or setting HyperTalk properties.

HyperTalk allows you to control the following features of a stack's card window:

- the current size of the card window
- the current location of the card window
- the type of title shown in the title bar of the card window
- the current position or scroll of the card within the card window

Card windows are resized automatically when the card is resized from either the Stack info dialog box or with HyperTalk. You can resize card windows independently of the card. Card windows are located within the global coordinates of the screen. Cards are located within the card window's local coordinates. The representation of the relationship between the location of the card window and a larger card is shown in Figure 2-2. You can scroll the contents of a card that is larger than the current card window with the `scroll` property. See the `location`, `rectangle`, and `scroll` properties in Chapter 12, "Properties," for more information about moving and sizing card windows and cards.

■ **Figure 2-2** Relationship between the location of a card and a card window



When you resize a card with the `rectangle` property its card window is repositioned in the middle of the current screen.

To find out how to refer to HyperCard card windows with the HyperTalk language, see Chapter 5, "HyperCard Elements."

HyperCard built-in external windows

HyperCard has several built-in external windows. They are the script editor window, the navigator palette, the Message Watcher window, and the Variable Watcher window. You can control the following properties of these windows with HyperTalk:

- the current font and style of text used in the script editor window
- the current location of the Message Watcher window
- the current location of the Variable Watcher window
- the current location of the navigatior palette
- the current size of the Variable Watcher window
- the position of the horizontal and vertical bars of the Variable Watcher window

For more information about the script editor window, the Message Watcher window, and the Variable Watcher window, see Chapter 3, "The Scripting Environment."

Menus

You can also control the behavior of HyperCard menus with HyperTalk commands. Menus are not, however, HyperCard objects either, because like windows, they do not have a script and can't respond to messages. (Messages and scripts are described later in this chapter.)

Menus are containers. They contain menu items. Menu items can send messages to HyperCard objects with menu messages. If a menu item has a menu message, the menu message is sent to the current card or specified object when the menu item is chosen.

HyperTalk allows you to perform the following actions on menus:

- create new menus in the HyperCard menu bar with the `create menu` command
- add menu items to menus with the `put` command
- change the behavior of menu items by modifying a menu item's menu message with the `menumsg` property
- change the font and style of menu item text with the `textFont` and `textStyle` properties

- assign Command-key equivalents to menu items with the `cmdChar` property
- assign check marks for menu items (to show they are chosen) with the `markChar` property
- determine the name of menus or menu items with the `name` property
- disable menus and menu items with the `disable` command
- enable menus and menu items with the `enable` command and `enabled` property
- delete menu items with the `delete` command (except in the Font, Tools, and Patterns menus)
- delete entire menus with the `delete` command (once deleted, they're gone, unless you recreate them)
- reset the HyperCard menu bar with the `reset menu` command (once reset, all custom menus created with the `create menu` command are deleted)

To find out how to refer to HyperCard menus and menu items with the HyperTalk language, see Chapter 5, "Referring to Objects, Menus, and Windows."

Chapter 3 **The Scripting Environment**

This chapter describes the environment for creating and editing the scripts of HyperCard objects. It also describes the built-in script debugger that is part of the scripting environment.

Getting to the script

You can get to an object's script through the Objects menu. The Objects menu has five object Info items, one for each of the five types of objects: the buttons and fields belonging to the current card and background, the current card, its background, and the stack the current card belongs to.

■ **Figure 3-1** The Objects menu

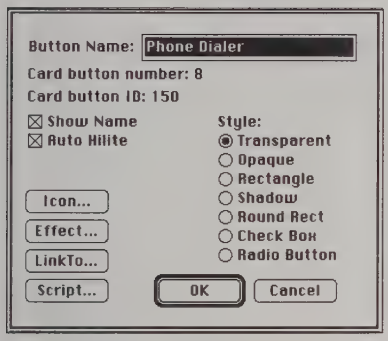
◆ *You must be at level 5:* The user level must be set to 5 for you to be able to look at scripts. To change the user level, choose Preferences from the Home menu and select Scripting on the Preferences card, or set the user level to 5 from the Message box with this statement:

```
set userLevel to 5
```

To edit the script of the current card, background, or stack, choose the appropriate Info menu item for the object whose script you want. This action brings up information about the object in an Info dialog box (see Figure 3-2). To open the object's script, click the Script button in that object's Info dialog box.

To get to the script of a button or field, first select the button or field (with the Button tool or Field tool), then choose the appropriate Info item from the Objects menu. It is not necessary to be working in the background to open the script of an existing background button or field. You must be working in the background, however, to create new background buttons and fields. (Working in the background may also make it easier to select background buttons and fields, because when you're in the background, HyperCard doesn't display card buttons and fields.)

■ **Figure 3-2** Button Info dialog box



◆ *Shortcuts:* To get to the Info dialog box of a button or field quickly, double-click the button or field with the Button or Field tool chosen.

To open a button or field script directly, hold down the Command-Option-Shift keys while you click anywhere inside the object's surrounding rectangle. To open the script of the current card, press Command-Option-C. To open the script of the current background press Command-Option-B. To open the script of the current stack, press Command-Option-S.

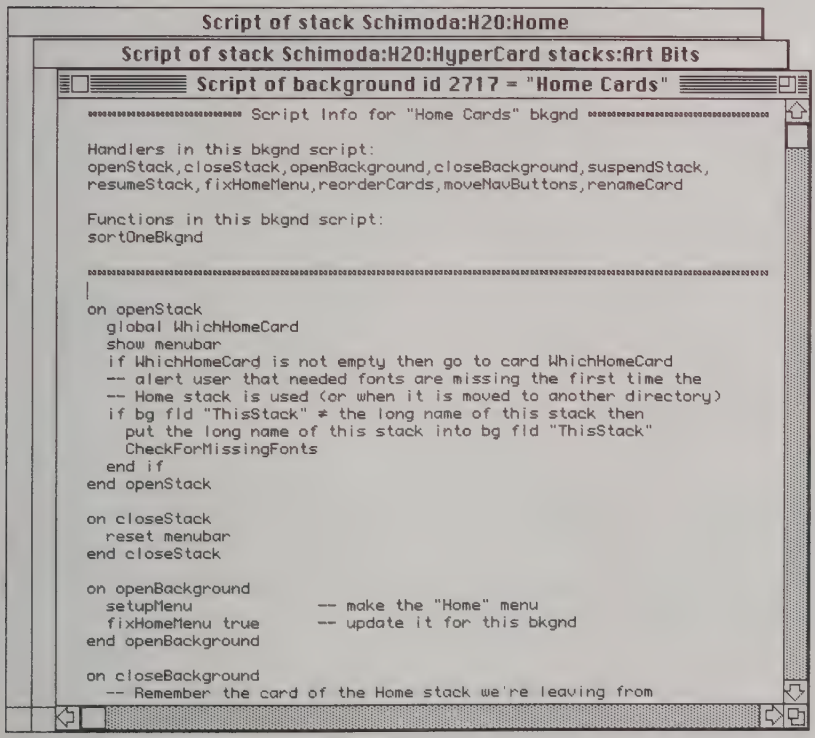
You can close a script by using the Close Script command in the File menu, by pressing Command-W, or by clicking the close box in the upper-left corner. You can save a script by using the Save Script command in the File menu or pressing Command-S. To close the script without saving changes, press Command-period.

- *Shortcut*: To save and close a script quickly, press the Enter key.

The script editor

The HyperCard **script editor** lets you create and modify handlers in an object's script. You can have more than one script open at a time. The scripts may be from within the same stack or from different stacks (see Figure 3-3). The number of possible open scripts depends on the available memory in the Macintosh. When you work on a script, you are working in a **script window.** You can resize script windows and drag them to any position on the screen.

■ **Figure 3-3** Script editor window


Script of stack Schimoda:H20:Home
Script of stack Schimoda:H20:HyperCard stacks:Art Bits
Script of background id 2717 = "Home Cards"
Script Info for "Home Cards" bkgnd
Handlers in this bkgnd script:
openStack,closeStack,openBackground,closeBackground,suspendStack,
resumeStack,fixHomeMenu,reorderCards,moveNavButtons,renameCard
Functions in this bkgnd script:
sortOneBkgnd
on openStack
global WhichHomeCard
show menubar
if WhichHomeCard is not empty then go to card WhichHomeCard
-- alert user that needed fonts are missing the first time the
-- Home stack is used (or when it is moved to another directory)
if bg fld "ThisStack" ≠ the long name of this stack then
put the long name of this stack into bg fld "ThisStack"
CheckForMissingFonts
end if
end openStack
on closeStack
reset menubar
end closeStack
on openBackground
setupMenu -- make the "Home" menu
fixHomeMenu true -- update it for this bkgnd
end openBackground
on closeBackground
-- Remember the card of the Home stack we're leaving from

- *Note:* The script editor is implemented as an external window; you can replace it with a custom script editor that you define as an external command. See Appendix A, "External Commands and Functions," for information on external commands and external windows; see the `scriptEditor` property in Chapter 12, "Properties," for how to change to a custom script editor.

While a script window is active, the script editor menu bar, which includes a menu called Script, is accessible (see Figure 3-4). The commands in the Script menu are described later in this chapter.

Figure 3-4 Script menu

When you have a script window open, you can still use the regular HyperCard menu bar by making a card window, rather than a script window, active. If any part of a card window is visible, you can make it active by clicking it. You can also use the Next Window command from the Go menu or press Command-L to bring a card window to the front.

Manipulating text

Many standard text editing features are available in the script window. You can use the arrow keys to move the text insertion point around in the script. If your script extends beyond the right border of the script window, you can scroll horizontally by using the scroll bar at the bottom of the script window. You can save the current script with Command-S and close it with Command-W. You can also print the current script with Command-P.

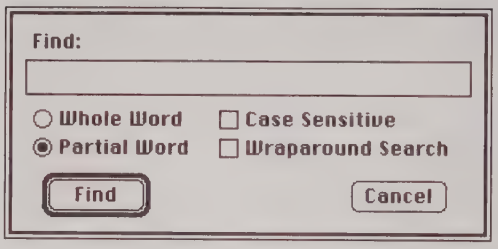
The mouse manipulates an I-beam pointer with which you can place an insertion point or select text. You can double-click to select a word or triple-click to select a whole line. You can perform cut, copy, and paste operations using Command-X, Command-C, and Command-V, respectively. The selection that you've cut or copied remains in the Clipboard until you cut or copy again, in case you want to paste the material more than once. You can also paste it into a field as regular text or on a card or background as Paint text. You can undo a cut, copy, clear, paste, or any typing operation with Command-Z or by choosing Undo from the Edit menu.

You can change the font, size, or style in which script text is displayed with the properties `scriptTextStyle`, `scriptTextFont`, and `scriptTextSize`. See Chapter 11, "Properties," for more information about these properties.

Searching for text

The Find command in the Script menu is different from the HyperCard Find command in the regular HyperCard Go menu. If you choose Find from the Script menu (or press Command-F) you get the dialog box shown in Figure 3-5. The script editor locates and selects the first occurrence, following the current insertion point, of a string you type into the Find field. You can search for a whole word or a partial word. If you don't check Case Sensitive, Find ignores whether the letters in the word are uppercase or lowercase. If you check Case Sensitive, the case of each letter in the search string and the target string must match exactly. If you check Wraparound Search, Find searches for a string starting from the current insertion point to the end of the script, and then wraps around to the beginning of the script to continue the search. If Wraparound Search isn't checked, Find locates a string only if it is after the current insertion point.

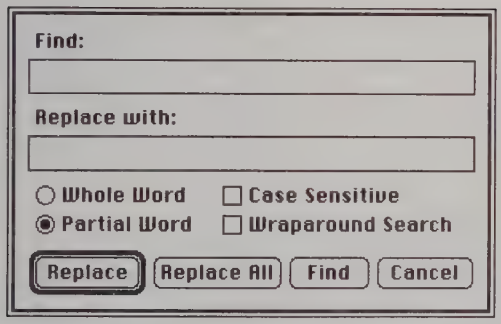
■ **Figure 3-5** Find dialog box



Replacing text

Choosing Replace in the Script menu or pressing Command-R brings up the dialog box shown in Figure 3-6. Replace locates and replaces the first occurrence, following the current insertion point, of a string you type into the Find field with the string you type into the "Replace with" field. For locating the string to be replaced, you have the same options as for the Find command: whole or partial word, case sensitive or not, and wraparound search or not. You can also specify that HyperCard replace the string, replace every occurrence of the string, or just find and highlight the string.

■ **Figure 3-6** Replace dialog box



Entering comments

You can put comments in your script by preceding the comment text with two hyphens (--). HyperCard ignores any text on a line after the double hyphen. If a comment wraps to the next line in a script, it must have a double-hyphen at the beginning of that line, too. You can insert a double-hyphen at the insertion point by choosing Comment from the Script menu, by pressing Command-hyphen, or by typing two hyphens.

You can remove the double-hyphen from a line with Command-equal sign or by choosing Uncomment from the Script menu. For the Uncomment command to work properly, you must place the insertion point next to one of the hyphens, or select any part of the line, as long as the selection includes the hyphens.

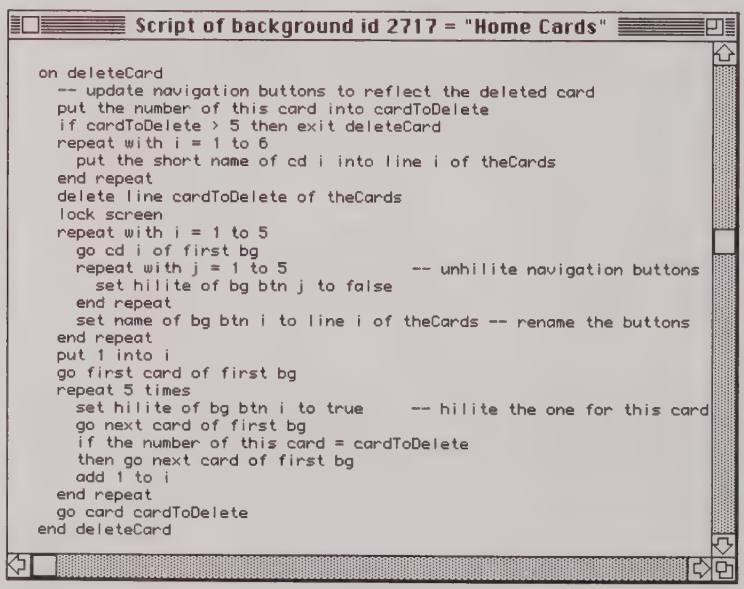
The double-hyphen is also useful when debugging or trying to improve the syntax in your scripts. You can precede statements in a handler with the double-hyphen to prevent them from executing. To have HyperCard ignore an entire handler, you need to add the double-hyphen to just the first line. This allows you to comment out a part of your script to see if that part was causing a problem.

Formatting scripts

The HyperCard script editor indents control structures for you. It automatically indents all of the lines inside a handler structure when you finish typing a statement and press the Tab key, press the Return key at the end of the last line, or close its script window. (See Figure 3-7.) When `if` and `repeat` structures are nested inside each other or within handlers, the lines are indented further. (You can't nest handler structures inside each other or any other structure.)

- *Error checking:* Automatic formatting provides some degree of error checking while you write a script: if you press the Tab key and the ending line in your handler isn't flush with the left margin of the script editor window, you probably left something out or made a syntax error in a HyperTalk command.

■ **Figure 3-7** Nested control structures


Script of background id 2717 = "Home Cards"
on deleteCard
-- update navigation buttons to reflect the deleted card
put the number of this card into cardToDelete
if cardToDelete > 5 then exit deleteCard
repeat with i = 1 to 6
put the short name of cd i into line i of theCards
end repeat
delete line cardToDelete of theCards
lock screen
repeat with i = 1 to 5
go cd i of first bg
repeat with j = 1 to 5 -- unhilite navigation buttons
set hilite of bg btn j to false
end repeat
set name of bg btn i to line i of theCards -- rename the buttons
end repeat
put 1 into i
go first card of first bg
repeat 5 times
set hilite of bg btn i to true -- hilite the one for this card
go next card of first bg
if the number of this card = cardToDelete
then go next card of first bg
add 1 to i
end repeat
go card cardToDelete
end deleteCard

Line length and script size

The script editor doesn't wrap lines that are too long to fit in a script window. Lines too long to fit in the script window simply extend out of sight. Line length is, however, limited to 255 characters. A single script cannot exceed 30,000 characters, including spaces, return characters, and other invisible characters. If you reach this limit, think about moving some of your handlers to another object, such as a hidden field or button, and send messages to it as required.

If you don't want statements to extend beyond the right boundary of the script window, you can break a single statement into multiple lines by pressing Option-Return where you want a line to break. This "soft" return appears in HyperCard scripts as a logical NOT symbol (¬). HyperCard treats lines broken in this way as single HyperTalk statements continuing to the next actual return character.

- *You can't break a literal:* You can't put a "soft" return inside a quoted literal expression. (Literals are described in Chapter 6.)

Script editor command summary

Table 3-1 is a summary of the script editor commands you can evoke from the keyboard.

■ **Table 3-1** Script editor command summary

Key press	Action
Command-A	Select all
Command-C	Copy selection to Clipboard
Command-D	Set or clear temporary checkpoint at selected line (for debugger)
Command-F	Display Find dialog box
Command-G	Find again
Command-H	Find the string currently selected elsewhere in the script
Command-L	Next window to front
Command-P	Print script or selection
Command-R	Display the Replace dialog box
Command-S	Save script
Command-T	Replace again
Command-V	Paste Clipboard contents at insertion point
Command-W	Close script
Command-X	Cut selection to Clipboard
Command-Z	Undo last operation
Command-hyphen	Comment selected line
Command-equal sign	Uncomment selected line
Command-period	Close script without saving changes
Tab	Format script
Option-click	Set or clear temporary checkpoint at selected line (for debugger)
Option-Return	Carry statement onto new line ("soft" return ¬)
Enter	Save changes and close script
Option–click close box	Close all open scripts

The debugger environment

This section describes the HyperTalk script debugger. The script debugger is integrated with the script editor to provide a set of easy-to-use debugging tools. You may find it easier to understand the features of the script debugger after learning more about HyperTalk, so you may want to skip this section and return to it later.

The HyperTalk script debugger has the following features:

- integration with the script editor for setting checkpoints within scripts
- debugging tools in the Debugger menu
- two additional XCMD windows (Message Watcher and Variable Watcher) for watching the progress of a script while it executes or while you step through the statements in a handler

In HyperTalk, a **checkpoint** serves the same purpose as a breakpoint in traditional development environments. You set a checkpoint at the location in a script at which you want to enter the debugger. You can also enter the debugger with the Command-Option-period key combination anytime while a script is executing. For example, you can get to the debugger by pressing Command-Option-period right after clicking a button that goes to another stack.

When HyperTalk enters the debugger, it pauses execution of the script, displays the script in a window, puts a box around the next line of the script to be executed, and displays a Debugger menu at the right end of the HyperCard menu bar. The Debugger menu, shown in Figure 3-8, has several menu items you can use to debug your scripts.

■ **Figure 3-8** The Debugger menu

You can step through the remaining statements in the script with Command-S or by choosing Step from the Debugger menu. Each step executes the statement with a box around it, then moves the highlight to the next statement in that handler.

You can step into the trail of subhandlers with Command-I or by choosing Step Into from the Debugger menu. Step Into allows you to follow execution among multiple handlers when one handler calls another. When you choose Step Into and a message is sent from the currently executing handler, you go to the location in the current object's script or any object's script in the message-passing hierarchy that has a handler for that message. That message handler becomes a subhandler to the originally executing handler. You can then continue to step through the subhandler until its completion. After completion of the subhandler and any of its subhandlers, you go back to the line in your original handler following the statement that sent the message to the first subhandler. Subhandlers could be in any object script within the current message-passing hierarchy. (See Chapter 4, "Handling Messages," for information about handlers calling handlers and the message-passing hierarchy.)

You can trace the current handler to completion by choosing Trace from the Debugger menu. When you select Trace, HyperCard executes each line in the current handler. Use Command-T or choose Trace Into in the Debugger menu to execute each line of the current handler including all of the subhandlers until the script's completion without having to manually step through the handlers. You can set the amount of time HyperCard waits between execution of the lines in a handler during a trace by choosing Trace Delay in the Debugger menu. The trace delay value can also be set with the global property `traceDelay`. See Chapter 12, "Properties," for information about the `traceDelay` property.

You can exit the debugger with Command-G or by choosing Go from the Debugger menu.

Setting checkpoints

You can set temporary and permanent checkpoints in a script. To set a temporary checkpoint in a HyperTalk script, set the insertion point anywhere in a line of a handler at which you want to enter the debugger and choose Set Checkpoint from the Script menu or press Command-D. You can also click anywhere in the line while pressing the Option key. A temporary check mark appears in the margin to the left of the chosen location. To remove a temporary checkpoint, perform any one of the previously mentioned operations on the line with the check mark. You can clear all the checkpoints in a script by clicking on any checkpoint in the script while pressing Shift-Option.

You can have up to 16 temporary checkpoints per script in a maximum of 32 scripts. Temporary checkpoints are not saved with the script when you quit HyperCard.

You set permanent checkpoints in a script by inserting the HyperTalk statement `debug checkpoint` anywhere within a handler. There is no practical limit to the amount of permanent checkponts in a script. Permanent checkpoints are permanent in that they are saved with the script—they can be removed by deleting the `debug checkpoint` statement.

Checkpoints are ignored at user levels less than five.

HyperTalk debugger windows

The debugger windows are named the Message Watcher and the Variable Watcher. You can display one or both of these windows in two ways. They can be called by HyperTalk commands in a script or the Message box, or they can be displayed by choosing Variable Watcher or Message Watcher from the Debugger menu while you're in the debugger environment.

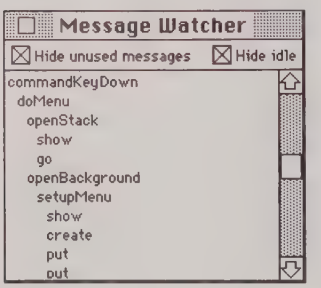
Message Watcher

The Message Watcher, shown in Figure 3-9, is an external window. It appears as a floating movable window that displays HyperTalk messages as they are sent. You can display the Message Watcher window with either of the HyperTalk statements:

```
show Message Watcher
set the visible of window "Message Watcher" to true
```

When you're in the debugger, you can also display the Message Watcher window by choosing Message Watcher from the Debugger menu.

■ **Figure 3-9** The Message Watcher window



The Message Watcher window is always in the front of the active window, which may be a card window or script window. (If you call the Message box, it appears in front of the Message Watcher.) You can close the Message Watcher window by clicking the close box or with either of these statements:

```
hide Message Watcher
set the visible of window "Message Watcher" to false
```

When you're in the debugger, you can close the window by choosing Message Watcher from the Debugger menu.

When "Hide unused messages" is checked, the Message Watcher displays only those messages that are handled by scripts. If "Hide unused messages" is not checked, you see all messages that are sent whether or not they are intercepted by a handler. Each message that isn't handled in the message-passing hierarchy is displayed in parentheses. For example, `(mouseWithin)` displayed in the list indicates that there was no handler for the `mouseWithin` message in the message-passing hierarchy.

When "Hide idle" is checked, the Message Watcher doesn't display idle messages. If "Hide idle" isn't checked, idle messages are displayed.

The Message Watcher stores the last 150 lines of messages. Older messages are removed as the new messages fill the buffer.

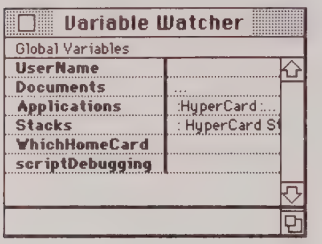
Variable Watcher

The Variable Watcher, shown in Figure 3-10, is an external window. It appears as a floating window that displays the HyperTalk global variables and local variables used by the current script. (Variables are explained in Chapter 6, "Values.") Variables are displayed in a two-column format with the variable name in the left column and the current values in the right column. You can display the Variable Watcher window with either of the HyperTalk statements:

```
show Variable Watcher
set the visible of window "Variable Watcher" to true
```

When you're in the debugger, you can also display the window by choosing Variable Watcher from the Debugger menu.

■ **Figure 3-10** The Variable Watcher window



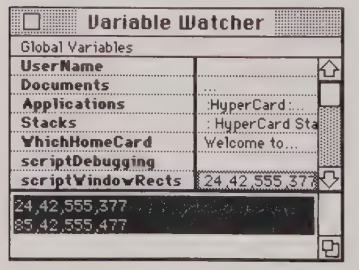
The Variable Watcher window is always in the front of the active window, which may be a card window or a script window. You can close the Variable Watcher window by clicking the close box or with either of these statements:

```
hide Variable Watcher
set the visible of window "Variable Watcher" to false
```

When you're in the debugger, you can close the window by choosing Message Watcher from the Debugger menu.

You can modify the values of global and local variables to see how they affect your script with the Variable Watcher. Drag the Variable Watcher size box to expand the window so that you can see the area below the thick horizontal line in the window. Click the variable name that you want to modify. The variable value is highlighted and placed in the field below the horizontal bar as shown in Figure 3-11. You can edit the variable value by typing into the field. When you have finished changing the value, press the Enter key. The value for that variable is changed.

■ **Figure 3-11** A selected variable in the Variable Watcher window



Custom message watcher and variable watcher XCMDs

The Message Watcher and Variable Watcher are implemented as external windows, so they can be replaced with custom external commands written in a programming language such as Pascal.

To replace either the Message Watcher or Variable Watcher window with your own custom tools, set the global properties `messageWatcher` or `variableWatcher` to the name of your tool with a HyperTalk statement like

```
set variableWatcher to "MyToolWindow"
```

For more information about creating custom message watcher and variable watcher external windows, see Appendix A.

Debugger command summary

Table 3-2 is a summary of the debugger commands you can evoke from the keyboard.

■ **Table 3-2** Debugger command summary

Key press	Action
Command-S	Step to the next line in current handler; step through any subhandler
Command-I	Step into and follow the path through any subhandlers
Command-T	Trace the handler and its subhandlers to completion. Same as Step Into, but user interaction isn't required
Command-G	Continue running the scrpt
Command-D	Set/Clear the current checkpoint
Command-A	Stop the handler
Command-period	Stop the handler and bring card window to the front
Command-Option-period	Open the debugger while a script is running

Chapter summary

Here is a summary of the material covered in this chapter:

- You can create and edit scripts with the HyperCard script editor.
- You can debug scripts with the built-in debugger.

Chapter 4 **Handling Messages**

This chapter explains how HyperCard objects send and receive messages and how HyperCard executes scripts.

The HyperCard environment

HyperCard provides the environment in which HyperTalk scripts execute. The HyperCard environment consists of objects connected by a message-passing hierarchy and the HyperTalk language through which they communicate.

Although you could write a stand-alone program in a single HyperTalk script, you would not be making use of the power and flexibility of the HyperCard environment. Instead, you use HyperTalk to define the ways in which objects interact with each other and with the user.

HyperCard is user oriented. When using HyperCard, the user opens and closes cards, reads and changes text in fields, draws pictures on cards, and so on. HyperCard constantly sends messages to objects in response to these actions (and the user's inactivity when doing nothing), and the objects in turn respond with other messages and other actions. The basic purpose of HyperTalk scripts is to enable objects to handle those messages and to specify succeeding actions by sending further messages.

Most of the time, scripts carry out specific actions for the user: setting properties of objects, going to other cards, and so on. HyperTalk can do automatically almost everything the user can do manually with the mouse and keyboard.

Sending messages

All HyperCard actions are initiated by messages sent to objects. Messages are sent to objects in four ways:

- An event (such as a mouse click or a key pressed on the keyboard) can cause HyperCard to send a system message.
- Handler statements (other than keywords) are sent as messages when a handler executes.
- HyperCard sends the contents of the Message box as a message when the user presses Return or Enter.
- HyperCard sometimes sends a message when it executes a command.

System messages

HyperCard sends system messages constantly in response to events in the Macintosh environment. For example, if you move the pointer so that it's over a button on the screen, as soon as the pointer enters the button's rectangle, HyperCard sends the message `mouseEnter` to the button. As long as the pointer remains inside the button rectangle, HyperCard continuously sends the message `mouseWithin` to the button. As soon as you move the pointer outside the button area, HyperCard sends the message `mouseLeave` to the button.

HyperCard sends other system messages when you press certain keys on the keyboard, close a field, select a menu item, or quit HyperCard. When you open a card, HyperCard sends the message `openCard` to the card itself; when you leave the card it sends `closeCard`. Similar messages are sent to cards when their backgrounds and stacks are opened and closed. If nothing at all is happening, HyperCard continuously sends the message `idle` to the current card.

One of the most commonly used messages is `mouseUp`. Buttons often contain handlers that respond to the `mouseUp` message; the `mouseUp` message is sent to a particular button when you click it. (HyperCard actually sends two messages to a button when it is clicked: `mouseDown` and `mouseUp`. The `mouseUp` message is sent only if you release the mouse button with the pointer over the same screen button it was over when you pressed it down.)

HyperCard also sends mouse messages to a locked field when you click it. If the field isn't locked, `mouseDown` and `mouseUp` aren't sent—the click opens the field for text editing and HyperCard sends the message `openField` to the field. (You can send mouse messages to an unlocked field, however, by holding down the Command key while you click the field.)

Clicking outside all buttons and fields sends `mouseDown` and `mouseUp` directly to the current card.

Chapter 8 describes all of the HyperCard system messages.

Statements as messages

When a handler executes, its statements are sent as messages, first to the object that contains that handler, then to succeeding objects in the message-passing hierarchy (described later in this chapter). When an object gets a message it can handle—that is, for which it has a handler in its script—the statements contained in the handler are in turn sent as messages. When all statements in the handler (and in any other handlers invoked along the way) have executed, the action stops.

Message box messages

When you type something into the Message box and press Return or Enter, HyperCard does one of these things: evaluates a valid expression and puts the result into the Message box, sends what you typed as a message to the current card, or sends a message to another destination if you use the `send` command. (See Chapter 6 for an explanation of evaluating expressions.)

You use `send` to direct a message to a specific object rather than sending it to the current card. `Send` is one of the HyperTalk keywords. If you try to use a keyword other than `send` in the Message box, HyperCard displays an error dialog box. The following list contains all of the HyperTalk keywords:

```
do          next
else        on
end         pass
exit        repeat
function    return
global      send
if          then
```

Chapter 9 explains the HyperTalk keywords.

Messages resulting from commands

HyperCard sometimes sends a system message to the current card while executing a command. For example, when you create a card with the New Card menu command, HyperCard sends the message `newCard` to the card as soon as it's created; when you delete a card it sends `deleteCard`. Similar messages are sent when other objects are created and deleted. These messages are among the results of commands executing, rather than commands themselves—they are like announcements of what is happening.

- *External commands can send messages:* Experienced programmers can write definitions for new commands in development languages such as Pascal, C, and 68000 assembly language. Such external commands act much like built-in HyperTalk commands. External commands can send messages to the current card when they execute. See Appendix A for information about external commands.

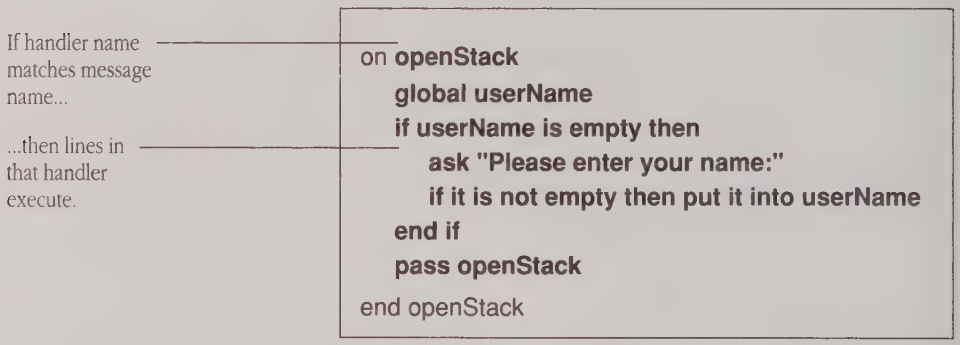
Receiving messages

As senders and receivers of messages, objects all work exactly the same way. Every object has a script, and the type of object makes no difference to the execution of its handlers.

- *How objects differ:* As elements of the HyperCard user interface, objects differ according to their function: buttons share a set of properties or characteristics that determine how they look and act; fields also share a set of properties, but it is different from the set of button properties. (See Chapter 12 for a description of object properties.)

When a message is sent to an object, HyperCard checks the object's script for a handler whose name—the second word on the first line of the handler—matches the message name—the first word of the message. If it finds a match, it executes each statement in the handler. (See Figure 4-1.) After the handler has run, the message is sent no further, unless it is explicitly passed with the `pass` keyword (discussed in Chapter 9).

■ **Figure 4-1** Handler that responds to message `openStack`


If handler name matches message name...
...then lines in that handler execute.
on openStack
global userName
if userName is empty then
ask "Please enter your name:"
if it is not empty then put it into userName
end if
pass openStack
end openStack

If the object has no handler for the message, the message passes to the next object in the hierarchy, and the process repeats. The message-passing hierarchy is explained in the next section.

If no object in the hierarchy has a handler matching a message name, HyperCard looks for a command by that name. Commands are like built-in handlers that cause some action to take place; `mouseUp` and most other system messages have no built-in handlers and cause no action. If a message gets all the way through the hierarchy and is not a system message or a command, HyperCard displays an error dialog box with the words `Can't understand` followed by the name of the message.

◆ *External commands can be in stacks:* External commands can exist in stack files, as well as in the HyperCard application itself. See Appendix A for information about external commands.

Message-passing hierarchy

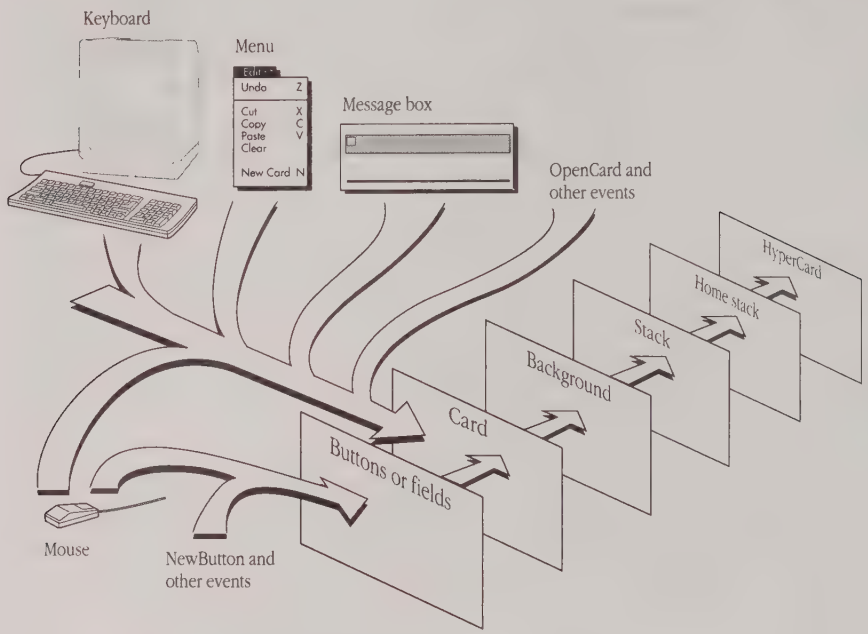
Messages are passed to objects according to a **message-passing hierarchy.** The message-passing hierarchy determines the path by which messages are passed from one object to another: buttons and fields are at the same level, followed (in order) by card, background, stack, the Home stack (the one stack that HyperCard requires), and HyperCard.

HyperCard allows you to add stacks to the message-passing hierarchy so you can use their scripts as shared-code libraries. More about the user-definable message-passing hierarchy is explained later in this chapter.

Where messages go

The position of an object in the message-passing hierarchy determines whether or not the object receives a given message, and where subsequent messages that the object sends go. Most system messages are initially sent by HyperCard to the current card, as shown in Figure 4-2.

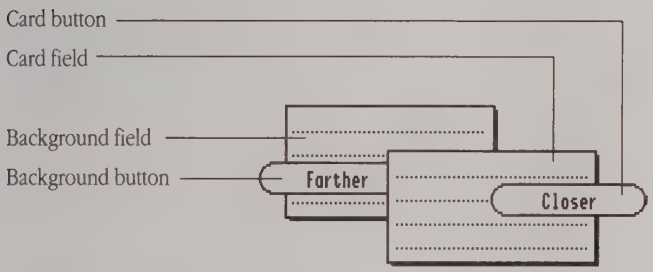
■ **Figure 4-2** Message-passing hierarchy



Messages to buttons and fields

Any mouse message (for example, `mouseEnter`) is sent initially to the topmost button or field, if there are any, under the pointer. Any buttons or fields that are layered farther under the one initially receiving the message are ignored. Figure 4-3 shows layered buttons and fields. If the topmost button or field doesn't have a handler for the mouse message, the message is passed to the current card.

■ **Figure 4-3** Layered buttons and fields



◆ *Background buttons and fields come before cards:* HyperCard first sends mouse messages to the topmost button or field under the pointer, whether the button or field belongs to the card or the background, before passing the message on to the card. Background buttons and fields, however, are always farther away than card buttons
and fields.

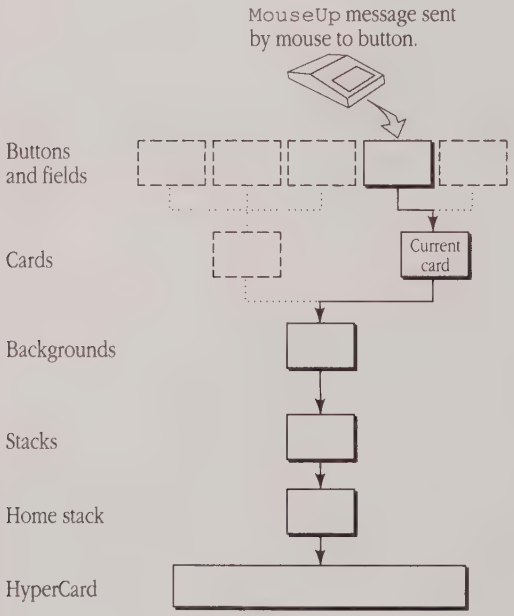
Other than mouse messages, the only system messages that are sent first to buttons are `newButton` and `deleteButton`; for fields they are `newField`, `deleteField`, `openField`, `closeField`, `enterInField`, and `returnInField`. The entry point in the hierarchy for all other system messages is the current card.

For a complete list of all system messages, see Chapter 8.

The current hierarchy

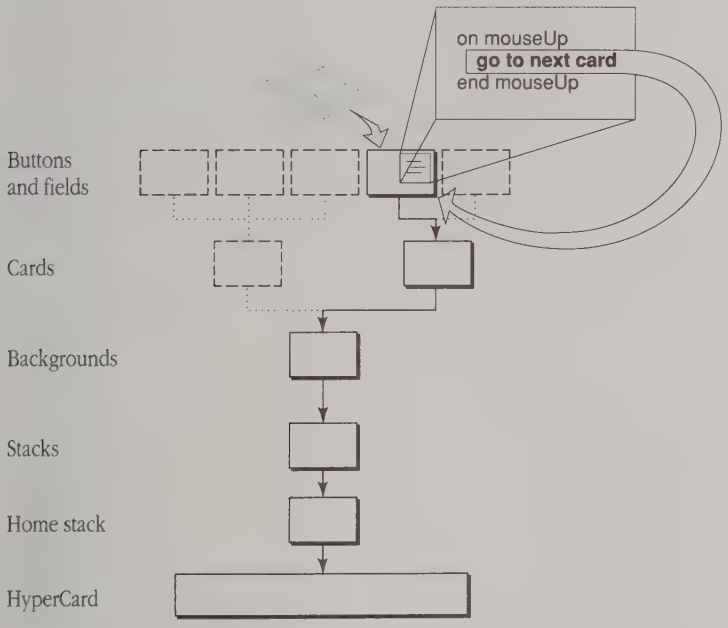
The current hierarchy consists of the buttons and fields belonging to the current card and its background, the card and background themselves, their stack, the Home stack, and HyperCard. System messages and those typed directly into the Message box always traverse the current hierarchy. Messages sent from executing handlers traverse the hierarchy in which their containing object belongs—in most cases, the current one. Figure 4-4 shows how a message traverses the current hierarchy.

■ **Figure 4-4** Message traversing current hierarchy



When a handler executes, HyperCard sends each statement as a message, unless it begins with a keyword. It sends the message first to the object containing that handler, as shown in Figure 4-5. If that object doesn't have a handler for the message, the message is passed down the object hierarchy; if none of the succeeding objects has a handler for it, the message ends up at HyperCard itself.

■ **Figure 4-5** Command sent as a message

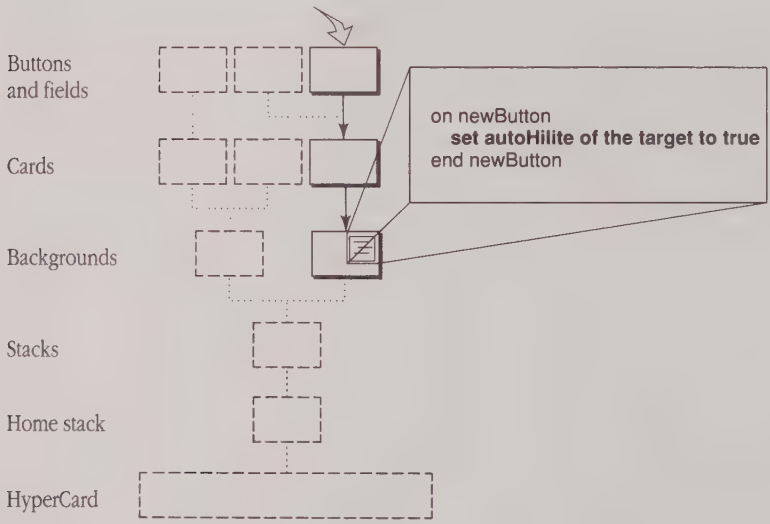


◆ *Function calls use the message-passing hierarchy:* Function calls work like messages in the way they traverse the object hierarchy. When you make a function call with the syntax that uses parentheses, HyperCard looks in the script of each object in the hierarchy for a matching function handler. If none is found, the function call is passed to HyperCard itself. See Chapter 11 for information about functions.

The target

The object to which the message is first sent is the **target.** If HyperCard finds a handler in the target that matches the message name, the handler's statements start executing. If, however, the target has no matching handler, the message is passed down the hierarchy. HyperCard may find a matching handler in another object, which then begins executing as shown in Figure 4-6.

■ **Figure 4-6** The target



The function `the target` returns the name of the original target, so that handlers in succeeding objects can determine where a message was originally sent. In Figure 4-6, although the executing handler is in the background script, `target`, used in the background handler, results in identifying the new button that originally received the system message `newButton`.

The user-defined hierarchy

HyperCard allows you to add stacks to the message-passing hierarchy, thereby extending the current hierarchy. Stacks added to the message-passing hierarchy act as shared code libraries. The code that they share is their script.

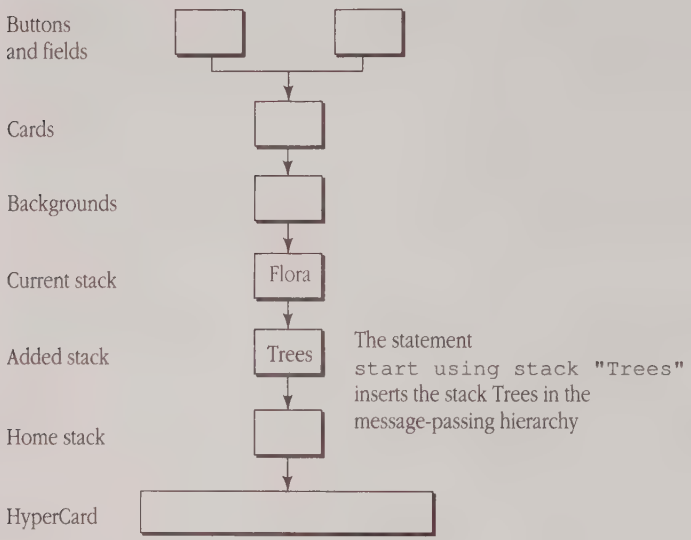
The message-passing hierarchy of HyperTalk always includes the Home stack and HyperCard itself. Scripts in any stack can call handlers in the script of the Home stack, and external commands in the resource fork of either the Home stack or HyperCard because they are always in the hierarchy.

When a stack is added to the hierarchy, it is inserted between the current stack and the Home stack, as shown in Figure 4-7. That stack's script and all of the stack's resources can now be shared with objects higher in that hierarchy.

To add a stack to the message-passing hierarchy, use the HyperTalk command `start using` in a handler like this:

```
on openStack
  start using stack "HD:Trees"
end openStack
```

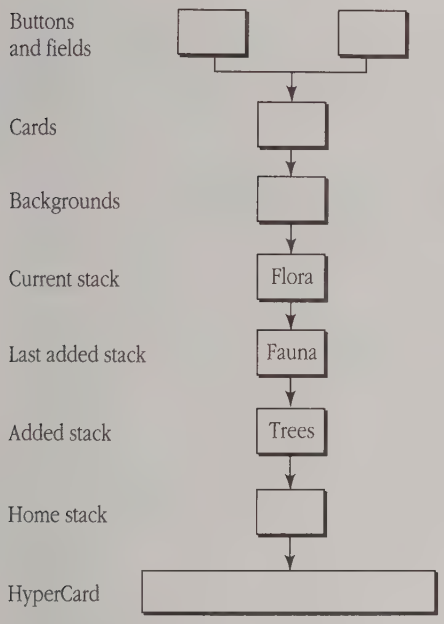
■ **Figure 4-7** One stack added to the message-passing hierarchy



After the `start using` command in the example handler is executed, the handlers in the script of the stack `Trees` and any of its external commands and resources are available for use by `Flora`.

Each additional stack that is added to the hierarchy is inserted after the current stack as shown in Figure 4-8. The maximum number of stacks you can have in a user-defined hierarchy is 10. If a stack that is already being used is used again with the `start using` command, it is moved in the hierarchy to the location just before the last stack that was inserted with the `start using` command.

■ **Figure 4-8** Two stacks added to the message-passing hierarchy



The current hierarchy described earlier in this chapter isn't changed when you create a user-defined hierarchy—it is extended with the new stacks added to the hierarchy. Messages still traverse the hierarchy in the same way: they go down from the buttons and fields belonging to the current card and its background, to the card and background, to their stack, to any added stacks, to the Home stack, and finally to HyperCard.

◆ *Note:* If you have a handler with the same name in more than one stack in a user-defined hierarchy, the handler highest in the hierarchy is executed when a message that calls that handler is sent. So if there's a handler for that message in the current stack, that's the one that gets executed. If there are handlers with the same name in two stacks that have been added to the hierarchy, the one in the most recently added stack is executed since it's higher in the hierarchy.

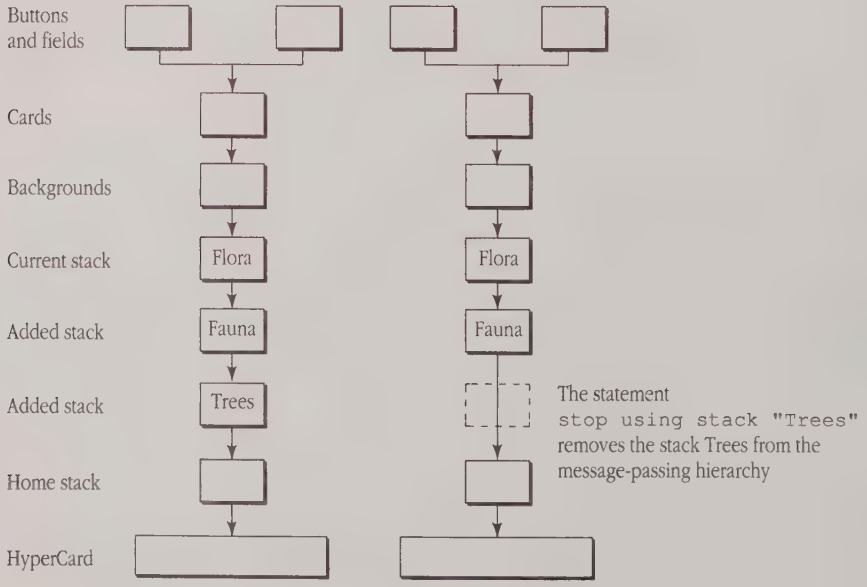
The names of the stacks in the current hierarchy are stored in the global property `stacksInUse` in the form of a return-separated list of the stacks in the order in which they receive messages. If you create a card field, and then use the statement `put the stacksInUse into card field "Myfield"` in the Message box, it would return the list of stack names in the card field you created. Each stack that was placed in the hierarchy with the `start using` command is on a separate line in the field. If no stacks were added to the hierarchy, `stacksInUse` returns empty.

You can remove a stack from the user-defined hierarchy with the command `stop using` in the following statement:

```
stop using stack "Trees"
```

The stack is removed from the hierarchy as show in Figure 4-9.

■ **Figure 4-9** Removing a stack from the message-passing hierarchy



The dynamic path

When a message is traversing the hierarchy of a card different from the current one, HyperCard inserts a dynamic path into the static path the message normally follows. The **static path** is the route defined by an object's own hierarchy. For example, a card passes messages to its own background, the background passes them to its own stack, and so on. When that hierarchy is not the one stemming from the current card (the one currently active), HyperCard passes messages through the current card's hierarchy as well—that's the **dynamic path.**

Examples of situations in which a message traverses a hierarchy different from the current one, invoking the dynamic path, are

- when an executing handler contains a command that takes you to another card (such as `go` or a command to create or delete the current card)
- when you use the `send` keyword to send a message to an object not in the current hierarchy

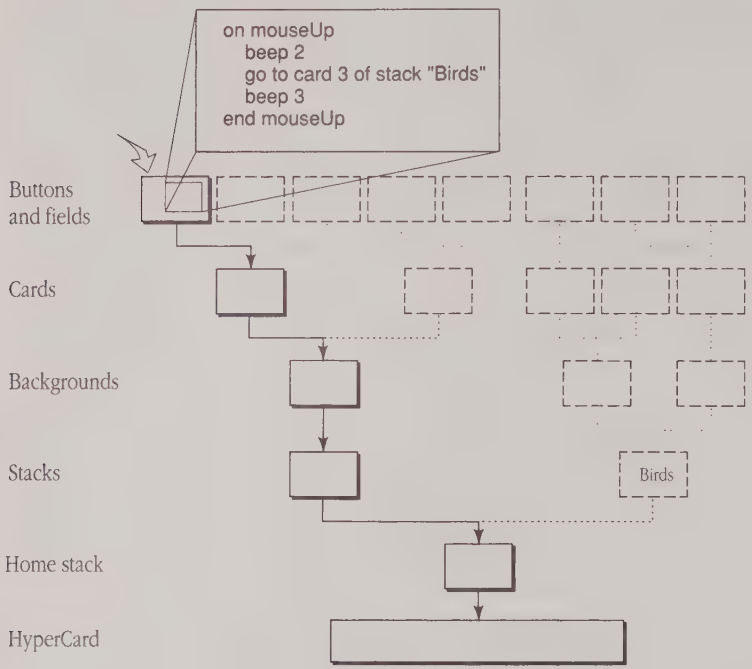
When any message that has not been received by a handler reaches the stack, HyperCard checks to see if the current card is in a different hierarchy. If so, HyperCard passes the message to the current card, and it traverses the current card, background, and stack, before it passes to the Home stack.

If any handler receives the message and passes it explicitly with the `pass` keyword, HyperCard does not invoke the dynamic path unless the current hierarchy is in a different stack from the static path. If either of the hierarchies is in the Home stack, the message is not passed again to the Home stack.

The `go` command and the dynamic path

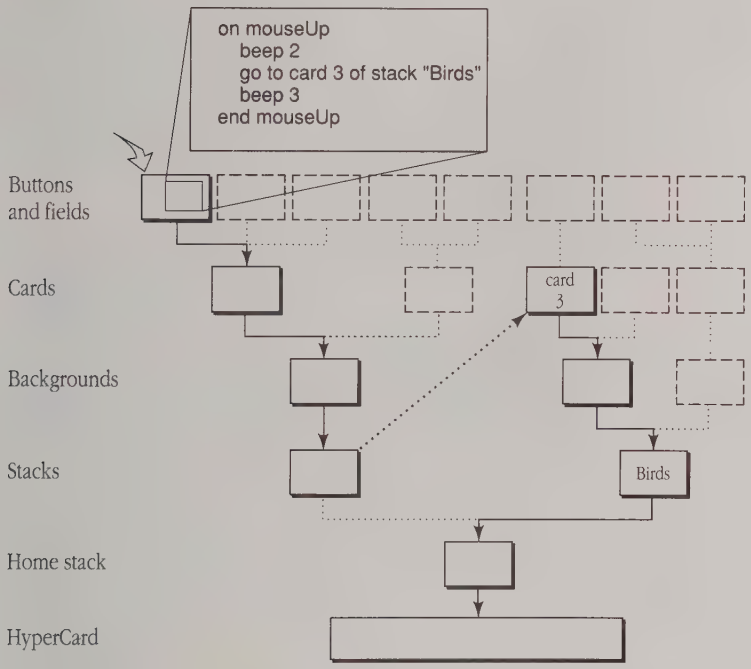
Figures 4-10 and 4-11 show how a handler containing a `go` command invokes the dynamic path.

■ **Figure 4-10** Static path before the `go` command executes



In Figure 4-10, the `mouseUp` handler executes the statement `beep 2`, which is sent as a message along the current hierarchy beginning with the button containing the handler. After the `go` command executes, the current card changes. Nonetheless, the button's `mouseUp` handler continues to execute, sending subsequent statements as messages through its own hierarchy, in this case the `beep 3` statement. In addition, however, HyperCard now sends messages to the card, background, and stack of the new current hierarchy, as shown in Figure 4-11.

■ **Figure 4-11** Dynamic path after the `go` command executes



The `send` keyword and the dynamic path

You can use the `send` keyword to direct a message to

- any object in the current stack
- any other stack on any disk or file server accessible to your Macintosh (but not any individual object in those stacks, unless you go to that stack first); the stack need not be in the current hierarchy
- HyperCard itself

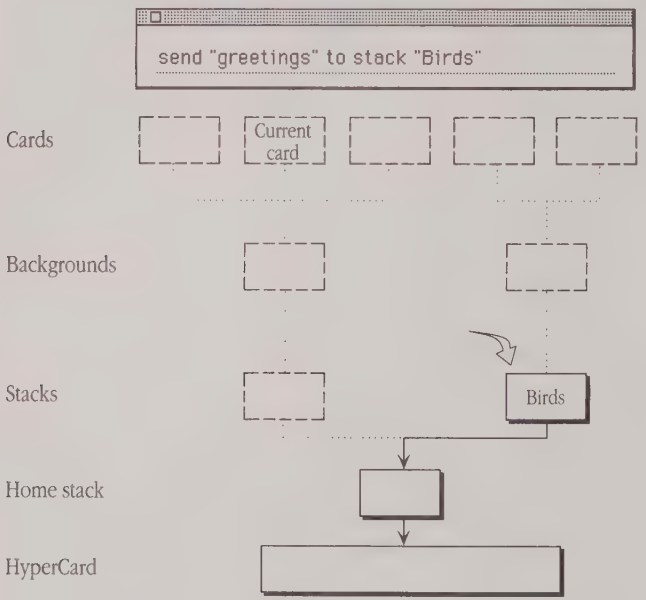
For example, you can type the following statement into the Message box:

```
send "greetings" to stack "Birds"
```

HyperCard looks in the script of the object to which the message is sent (in this case, `stack "Birds"`) for a matching handler, just as if it were in the current hierarchy. If the matching handler isn't found (in this case, a handler named `greetings`), the message goes down the hierarchy stemming from the object to which it was sent (that is, from `stack "Birds"`). If the target of the `send` command is an object other than the current one, HyperCard invokes the dynamic path.

Figure 4-12 shows the path of a message directed with the `send` keyword.

■ **Figure 4-12** Using the `send` keyword



The executing handler, the one currently in control, need not be in the hierarchy belonging to the current card. Which handler has control is determined solely by which object receives a message.

For details about the `send` keyword, see Chapter 9.

Handlers calling handlers

When a handler executes, HyperCard sends each statement as a message first to the object containing the executing handler. So other handlers in the same script, as well as those in any other script lower in the message-passing hierarchy, can be used as subroutines. A handler can also call itself, which is known as **recursion.**

Subroutine calls

You can use handlers in HyperCard the way you use procedures or subroutines in other languages. You invoke a subroutine call in HyperTalk by executing a statement that begins with the name of a handler. That name is sent as a message, first to the object that contains the executing handler, then along the current object hierarchy.

You can include a subroutine in a script by writing a handler in the same script (or any other script lower in the object hierarchy) with whatever name you'd like to call it by. In the following example, the handler `greetings` is defined in the same script as the one from which the message `greetings` is sent:

```
on mouseUp
  greetings
end mouseUp

on greetings
  Put "You've just been drafted!" into the Message box
end greetings
```

When HyperCard executes the statement consisting of the subroutine handler name, and a match is found between the name and its handler, control passes to the subroutine handler. After it has finished executing, control passes back to the calling handler. But it's entirely possible for the subroutine handler to issue a similar message, beginning execution of a third handler. The third handler must finish executing before control passes back to the second handler, which must finish executing before control passes back to the first. The execution of a handler that has invoked another handler is suspended until the handler it has called finishes executing.

- *Stopping execution:* A handler can avoid giving control back to pending handlers by executing the `exit to HyperCard` keyword statement. You can interrupt an executing handler at any time (and bypass pending handlers) by pressing Command-period.

Any handler can act as a subroutine for any other handler. The called handler either has to be in the same script or in a script lower in the object hierarchy. However, you can also use the `send` keyword to send the message (the subroutine handler name) directly to the object that contains the handler. (See Chapter 9 for details on using `send`.) Generally, handlers that act as subroutines are placed in the same script as the handlers that call them.

△ **Important:** Handlers can't be nested. That is, they can't be defined with one inside another—a handler definition must not appear between the `on` statement and the `end` statement of another handler. △

Recursion

If you need to repeat an operation over and over, you can have a handler call itself. In the following example, the handler `decrement` subtracts 1 from a number in the Message box until the number is reduced to 1 (a number must be in the Message box before you call the handler). To do the subtraction, the handler summons itself:

```
on decrement
   subtract 1 from the message box
   if the value of the message box > 1 then decrement
end decrement
```

Generally, subroutine calls and recursion don't cause any problems. In fact, they are natural consequences of the good programming technique of separating scripts into functional units. However, HyperCard has a limit on the number of pending handlers. The actual number depends on the complexity of the handlers and other factors. It doesn't matter whether a handler is invoking itself or another handler—either type of invocation causes another level of pending execution.

In particular, watch out for endless recursion as in the following handler (if it were in a stack script or the script of every card):

```
on openCard
  go to next card
end openCard
```

The `go next card` command results in an `openCard` message, so the handler recurses again and again, and you get an error dialog box. The limit on the number of iterations depends on the complexity of the script and the amount of RAM available.

Using the hierarchy

Where you place a handler in the hierarchy determines when it will be called. All objects that are higher in the hierarchy can call handlers in objects lower in the hierarchy. Lower objects can't call handlers in higher objects unless they use the `send` keyword. Messages that are sent when a statement in a handler executes always go first to the object containing the executing handler. Then they traverse the hierarchy stemming from that object until they find a matching handler or reach HyperCard itself. Therefore, the farther down the hierarchy a handler is placed, the greater the number of objects that can pass messages to it.

Sharing handlers

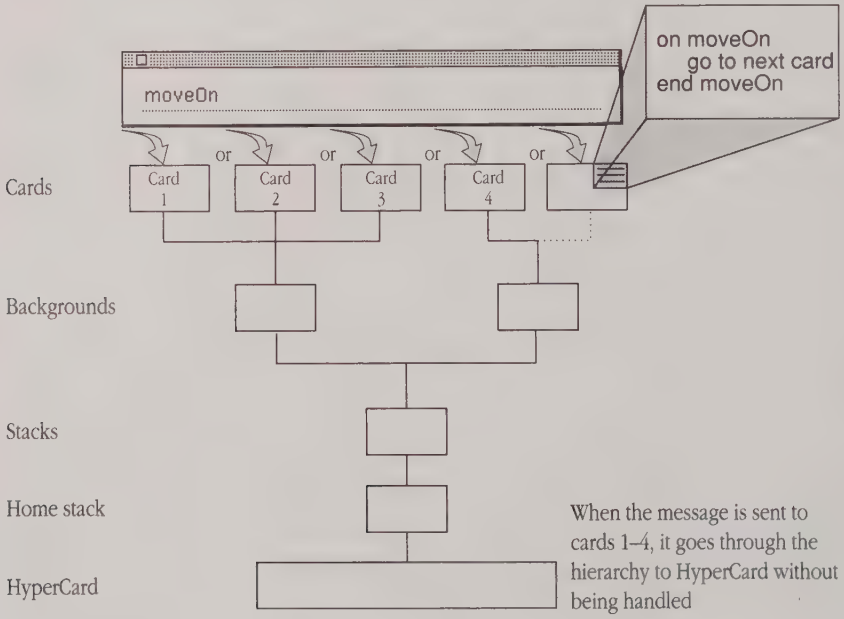
In effect, every object has access to the handlers of all the objects lower than it in the hierarchy, which also includes the handlers in stacks put into the message-passing hierarchy with the `start using` command. If you want every card in a stack to have a certain capability (that is, to respond to a certain message), you put the appropriate handler in the stack script. Every card can use the handler by passing the message down to the stack.

Figures 4-13 and 4-14 show the effect of placing a handler at different positions in the hierarchy. The example handler responds to the message `moveOn` (the message name is for example only). The handler takes you to the next card:

```
on moveOn
  go to next card
end moveOn
```

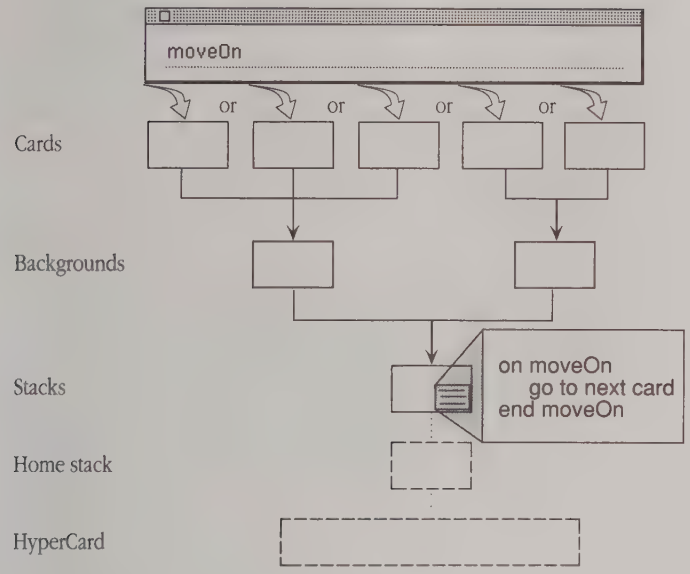
You can place the handler in the script of one card, as in Figure 4-13. Then, if you send `moveOn` from the Message box, you invoke the handler and go to the next card only if the current card is the one with the handler. If the current card is not the one with the handler, however, the `moveOn` message has no effect.

■ **Figure 4-13** Handler in card script



In Figure 4-14, the handler is in the script of the stack, so the handler is invoked by sending `moveOn` to any card in the stack.

■ **Figure 4-14** Handler in stack script



Intercepting messages

You can also make any card you want an exception in the way it responds to a given message, without affecting the other cards in the stack, by putting a special handler for the message in that card's script: you write two different handlers with the same message name—one in the stack script and one in the card script. Then, for that same message, if the message comes through that particular card, the card's handler runs; from any other card, the stack's handler runs.

For instance, in the previous example, putting the handler in the stack script caused the message `moveOn` to take you to the next card from any card in the stack:

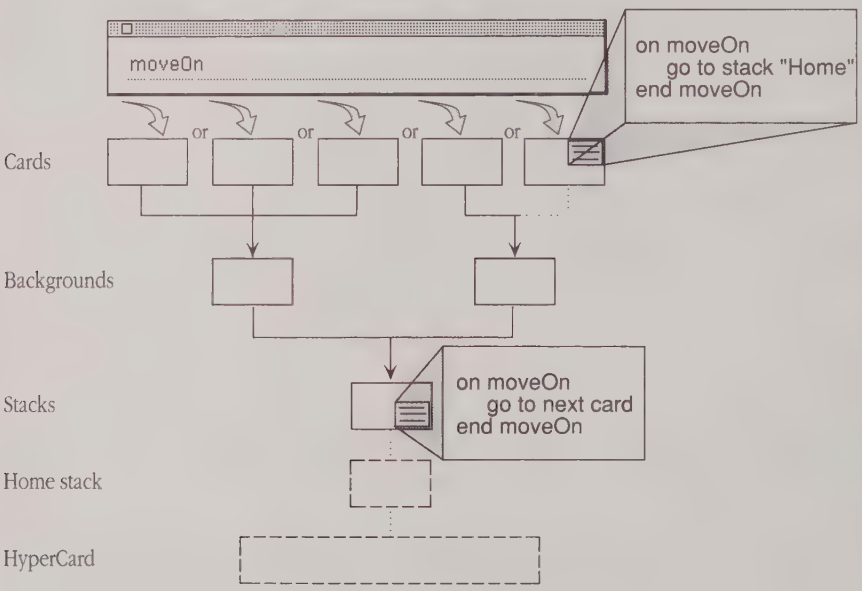
```
on moveOn
  go to next card
end moveOn
```

But if you want the last card in the stack to be an exception, from which the message `moveOn` takes you back to the Home card, put the following handler in the last card's script:

```
on moveOn
  go to stack "home"
end moveOn
```

Figure 4-15 illustrates this example of one object intercepting a message.

■ **Figure 4-15** Intercepting a message



- *A handler can intercept a HyperTalk command:* In the same way that you can give one card a unique way of handling a message that would ordinarily be handled in the background or stack script, you can write a handler with the same name as a HyperTalk command and place it anywhere in the hierarchy. But remember that your handler is the one that will ordinarily run in response to the command message, not the built-in one in HyperCard. HyperTalk functions can be redefined in a similar manner, and the same warning applies.

Parameter passing

When a HyperTalk message is sent, the first word is the message name. For example, in the message

```
searchScript "WildCard","Help"
```

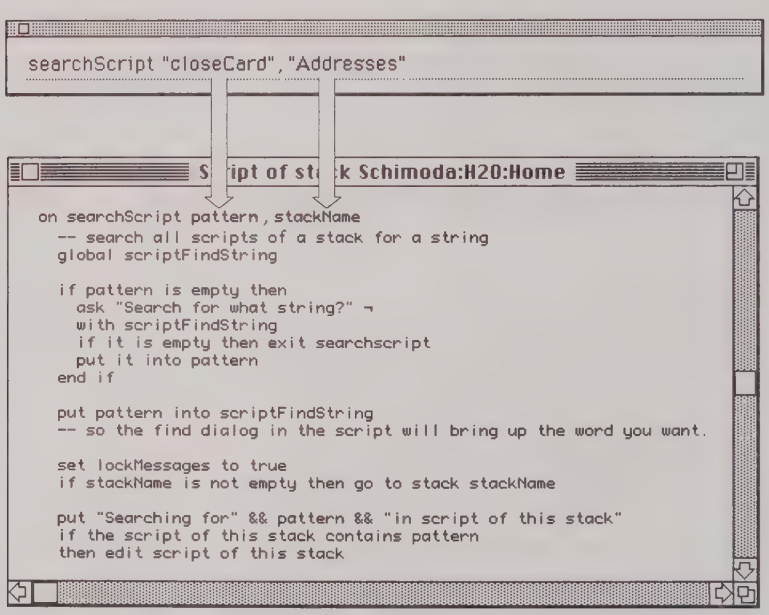
the message name is `searchScript`. Any other words (or characters) are the **parameters.** In the example, the parameters are `"WildCard"` and `"Help"`. Each receiving object in the hierarchy looks for a message handler with a matching name. If the object finds a matching handler, the parameters are passed into that handler.

Parameters are passed into handlers as a list of comma-separated expressions. (Chapter 7 describes expressions.) These expressions are evaluated before the message is sent by the current object and, when the message is received by the receiving object, placed into a list of comma-separated **parameter variables** appearing on the first line of the matching handler definition. (See Figure 4-16.) That is, parameters are passed by value into handlers. In the `searchScript` handler example shown in Figure 4-16, the parameter variables `pattern` and `stackName` are replaced by the parameter values `"WildCard"` and `"Help"`.

Parameter variables are local variables of the handler in which they appear. Parameter variables are also called **formal parameters,** to contrast them with the **actual parameters,** which are the parameter values passed to them.

- *Function handler parameters:* HyperCard passes parameters into function handlers and message handlers in the same way, except that the syntax of the function call requires the parameters to be placed between parentheses. Placement of the parameter variables on the first line of function handlers is identical to that of message handlers.

■ **Figure 4-16** Parameter passing


searchScript "closeCard", "Addresses"
Schimoda:H20:Home
on searchScript pattern, stackName
-- search all scripts of a stack for a string
global scriptFindString
if pattern is empty then
ask "Search for what string?" ¬
with scriptFindString
if it is empty then exit searchscript
put it into pattern
end if
put pattern into scriptFindString
-- so the find dialog in the script will bring up the word you want.
set lockMessages to true
if stackName is not empty then go to stack stackName
put "Searching for" && pattern && "in script of this stack"
if the script of this stack contains pattern
then edit script of this stack

The value of the first expression in the message is placed into the first parameter variable in the handler, the value of the second expression into the second parameter variable, and so on. If there are more expressions in the message's parameter list than there are parameter variables in the first line of the handler, the extra parameters are ignored. If there are more parameter variables than parameters, the extra parameter variables are given an empty value (equal to a string of zero length).

- *Passing parameters to redefined commands:* HyperTalk command parameters are often more complex than a comma-separated list of expressions. Some built-in commands take parameters to which user-written handlers have no access. So, if you redefine a command, you may not be able to pass all of the parameters to your handler.

Chapter summary

Here is a summary of the material covered in this chapter:

- The HyperCard environment consists of objects related to each other in a hierarchy using HyperTalk as the means of communicating.
- Messages sent to objects initiate all HyperCard actions.
- Messages are generated by system events, executing handlers, statements typed into the Message box, and the execution of some commands.
- When an object receives a message, HyperCard tries to match the message name with a handler in the object's script; if it finds a match, it executes the handler; otherwise it passes the message to the next object.
- The object hierarchy determines how messages are passed from one object to another.
- You can modify the message-passing hierarchy to use stacks as shared-code libraries.
- You can send a message directly to any object in the current stack, to another stack, or to HyperCard using the `send` keyword.
- A handler can initiate execution of another handler as a subroutine call.
- Every object can use the handlers of objects lower than it in the hierarchy by passing messages; conversely, an object can intercept a message to perform a different action.
- The values of a series of expressions following the first word of a message statement are passed to variables in the first line of the receiving handler.

Chapter 5 Referring to Objects, Menus, and Windows

This chapter explains how to refer to HyperCard objects, menus, and windows.

A HyperCard object has three characteristics:

- It can send and receive messages.
- It has properties, which are its defining characteristics, and one of those properties is its script.
- It has a visible representation on the Macintosh screen (although the object need not always be visible).

HyperCard menus and windows share many of the characteristics of HyperCard objects, except they don't have the script property.

You refer to an object when you use the `go` keyword (to go to a particular card, background, or stack) or the `send` command (to send a message to a particular object), and when you want to manipulate an object's properties. Fields are unique because they are HyperCard objects and are also sources of values (described in Chapter 6).

You can think of HyperCard itself as an object, because it can send and receive messages and it has global properties, including a "script" of built-in handlers or commands. When this guide talks about objects, however, it usually refers to buttons, fields, cards, backgrounds, and stacks.

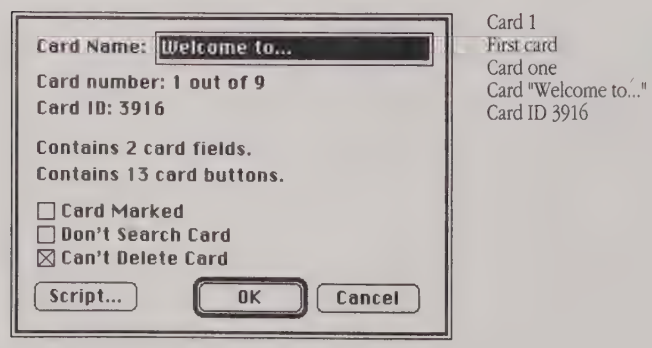
Names, numbers, and IDs

You refer to objects using **object descriptors.** An object descriptor is formed by combining a generic name with its specific designation. Generic names are `stack`, `card` (abbreviated `cd`), `background` (abbreviated `bkgnd` or `bg`), `button` (abbreviated `btn`), or `field` (abbreviated `fld`).

HyperCard considers all references to buttons to be card buttons and all references to fields to be background fields unless you specify otherwise. For example, `button "buttonName"` and `card button "buttonName"` both refer to the card button and `field "fieldName"` refers to the same field as `background field "fieldName"`. To refer to background buttons, you must include background in the descriptor; for example, `bkgnd button "buttonName"`. Similarly, if you want to refer to card fields you must include `card`; for example, `card field "fieldName"`.

The only specific designation of a stack is its name. (See "Identifying a Stack," later in this chapter.) The specific designation of all other objects (buttons, fields, backgrounds, and cards) can be the objects's name, number, or ID number. The unambiguous form of an object descriptor begins with an object's generic name, immediately followed by its particular name, number, or ID number. (See Figure 5-1.)

■ **Figure 5-1** Card Info dialog box and descriptors for the same card



- *Descriptor phrasing:* Be careful to phrase descriptors so that they mean what you intend. For example, using a descriptor such as `card field id 7`, you could mean that the name of the card is in the background field with ID number 7, or you could be referring to the card field with ID number 7. HyperCard assumes that you're referring to the card field. If you want HyperCard to get the card name from the background field, enclose its descriptor in parentheses:

```
card (field id 7)
```

Object names

Names are optional for cards, backgrounds, buttons, and fields. You assign a name for any of these objects by typing into the Name box in the object's Info dialog box, which appears when you choose the object's Info command from the Objects menu.

Object names can include any characters, even spaces. When you use a name (`background button "belly"`) in a statement, put quotation marks around the object name to ensure that HyperCard recognizes it as a literal and doesn't look for a variable by that name.

Names are not optional for stacks. You must provide a name for each new stack you create. A stack name must be a valid Macintosh filename.

- *Be careful with names:* It's difficult to manipulate a name that extends out of the naming window although you can scroll it left and right (and up and down if it has more than one line) by dragging. It's also difficult to refer by name to an object if you put a double quotation mark in its name. Also, if you use numbers for an object's name, HyperCard could misinterpret the name: it takes `card "1812"` to mean a card whose *number*, rather than name, is 1812.)

The name of an object is one of its properties. (See Chapter 12 for an explanation of properties and a description of the `name` property.)

The `name` property of an object has three forms—`long`, `abbreviated`, and `short`. The long name of an object includes the type of object, its name, its enclosing object (either a card or background), and the full pathname of its stack:

```
card button "Rolo" of card "Home" of stack "MyHardDisk:Home"
```

The abbreviated form includes the type of object and its name:

```
card button "Rolo"
```

The short form includes just the name:

```
"Rolo"
```

Object ID numbers

HyperCard generates an object ID number for each object within a stack. This number is unique for that type of object within its enclosing object. For example, each button (the type of object) on a card (the enclosing object) has a different ID number. Object ID numbers never change and, if an object is deleted, its number is not reassigned to a newly created object (until the HyperCard object limit, listed in Appendix G, has been reached). An object's ID number is its generic name, followed by the word `ID` (in uppercase or lowercase), followed by an integer (for example, `card id 5734`).

- *The ID number of a copied object is different:* If you copy an object and paste it into a different enclosing object, the copy is then a different object from the original, and it has a different ID number. For example, if you copy a card and paste it into a different stack, the ID number of the pasted card is different from the ID number of the card you copied. Therefore, you can't assume that you have "moved" the card when you copy it, paste it, and delete the original—a button that took you to the original will probably not take you to the copy.

Because ID numbers are unique and unchanging for all objects within a stack, HyperCard uses them internally to identify objects (for example, to identify the target of a `go` command generated with the LinkTo feature in the Button Info dialog box). HyperCard can generally find objects faster if they are identified by ID number. If you ask for the name of an object that has no name (`put the name of last card`), HyperCard returns its ID number.

The ID of an object is one of its properties. The `ID` property of an object has three forms which are similar to the three forms of the name, and which are differentiated by the same adjectives—`long`, `abbreviated`, and `short`. The long ID of an object includes the type of object, its ID number, its enclosing object if necessary, and the full pathname of its stack:

```
card id 2590 of stack "HyperDisk:HyperCard:Stacks:Home"
```

The abbreviated form includes the type of object and its ID number:

```
card id 2590
```

The short form includes just the ID number:

```
2590
```

All objects except stacks always have ID numbers; stacks never have ID numbers.

Object numbers

Buttons, fields, cards, and backgrounds always have numbers by which you can refer to them. An object's number represents its position within its enclosing object: buttons and fields are ordered within a card or background; cards and backgrounds are ordered within their stack. For objects with numbers one through ten, there are three ways to express an object's number: use an integer following its generic name (`card 2`), use one of the numeric constants `one` through `ten` following its generic name (`card two`), or use one of the ordinal constants `first` through `tenth` preceding its generic name (`second card`). For objects with numbers higher than ten, you have to use the integer value.

Object numbers are contiguous from one through the number of currently existing objects within the enclosing object: card buttons and card fields within their card; background buttons and background fields within their background; cards within their stack (not their background); and backgrounds within their stack. If you delete an object, its number is reassigned to the object following it in order, and so on for the succeeding objects as well.

Special ordinals

In addition to the ordinal constants `first` through `tenth`, HyperTalk has three special ordinals: `middle`, `last`, and `any`. The values of the special ordinals are resolved according to the number of objects in the set. `Middle` resolves to half the number of objects (rounded down to the nearest integer) plus 1. `Last` resolves to the number of objects. `Any` resolves to a random number between 1 and the number of objects. (The special ordinals also work with chunk expressions, which are described in Chapter 7.)

Object numbers and tab order

The sequence of object numbers determines tab order for fields: you can move from field to field within a background and card using the Tab key—it moves from the lowest numbered field to the highest through the background fields first, then the card fields. The sequence also determines which button or field gets a message when several are layered on top of each other (the highest numbered one is closest and gets the message), and it determines which card or background is referred to by the special descriptors `next` or `previous` (see the next section).

- *Reassigning object numbers:* You can reassign object numbers of buttons and fields with the Bring Closer and Send Farther menu commands. See the *HyperCard Reference* for details.

Special object descriptors

You can use the special descriptor `this` to refer to the current card, background, or stack. For example,

```
put the id of this card into whereFound
```

You can't use `this` with buttons or fields.

You can refer to the card or background preceding the current one, within the stack, as `previous`, which can be abbreviated `prev`. Similarly, you can refer to the card or background following the current one as `next`. For example,

```
go to next background
```

You can refer to the card that was current immediately prior to the current one as `recent`.

You use `me` within a script to specify the object containing the currently executing handler. For example, this statement in a field script would put the value of the `textHeight` property of the field into the variable `height`:

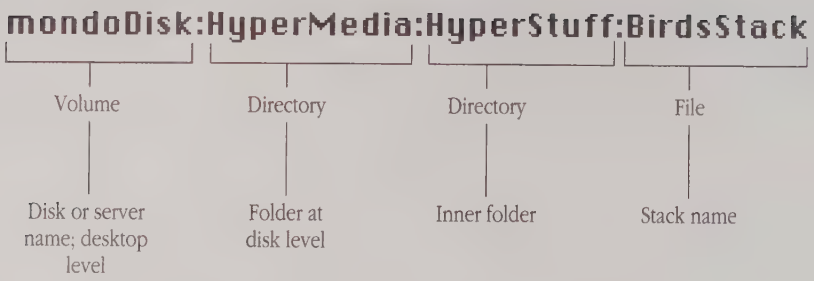
```
put the textHeight of me into height
```

Identifying a stack

A stack is a HyperCard document. In some cases when you're writing a script or using the Message box, you can refer to a stack by its name alone. To do that, the stack must be in the current folder, in the folder containing the Help stacks, or on the current disk or server (and not in a folder). When the stack is located anywhere else, you must let HyperCard know the full pathname by which it can find the stack.

A full pathname is a concatenation of the volume name, directory name or names, and filename, separated by colons. The volume name is the name of the disk or server containing the stack. The directory names are the names of all the folders, if any, that HyperCard has to open to get to the stack. (HyperCard sometimes might have to open several folders because folders may contain other folders to any depth.) The filename is the stack name. Figure 5-2 shows the structure of a pathname.

■ **Figure 5-2** A pathname



The only unambiguous way to refer to a stack in a script or in the Message box is the word `stack` followed by its name in quotation marks. When you refer to a stack, you can use the full pathname to specify the stack's exact location:

```
go to stack "myDisk:myFolder:myStack"
```

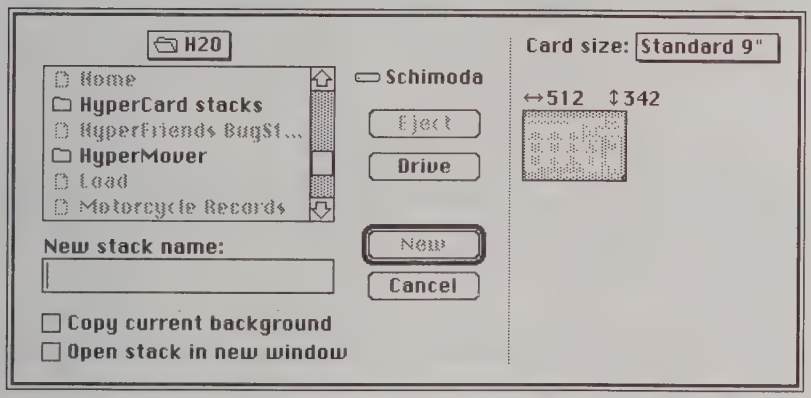
You can also type the full pathname in the Stack Search Paths card of the Home stack. If HyperCard can't find a stack you request, it displays a dialog box that allows you to click your way through the directories until you reach the stack. HyperCard notes your path and, once you've found the stack, automatically records its full pathname on the Search Path card in the Home stack.

- *Ambiguous stack descriptors:* HyperCard tries to derive a proper stack name from an ambiguous expression in a place where it expects a stack descriptor, but it cannot always succeed. In that case, HyperCard displays the directory dialog box to allow the user to find the stack file.

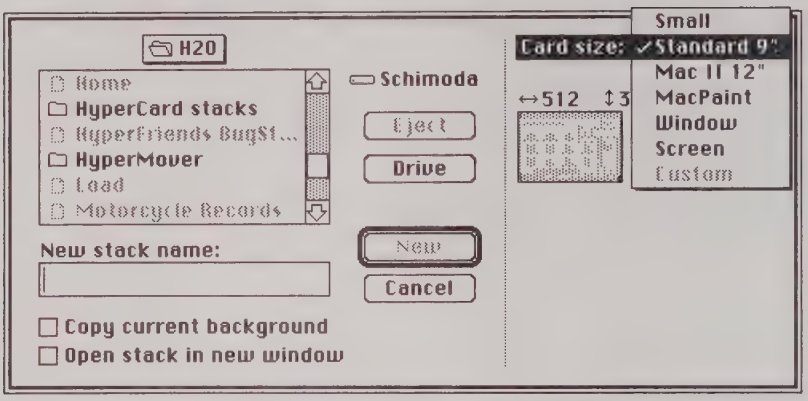
Naming a stack

You must name a stack when you create it. (For all other objects, names are optional.) You create a stack with the New Stack command in the File menu or with the `create stack` command. (See Chapter 10 for more information about the `create stack` command.) When you use the New Stack command, a dialog box appears in which you type the name for the new stack. (See Figure 5-3.) You can also select the card size for your stack by either dragging the rectangle on the right side of the dialog box, or by selecting one of the preset sizes available from the pop-up menu in the upper-right corner, as shown in Figure 5-4.

■ **Figure 5-3** New Stack dialog box



■ **Figure 5-4** New Stack Card size pop-up menu



Combining object descriptors

To refer to objects within a stack, you combine object descriptors using either of the prepositions `of` or `in` between an object descriptor and that of its enclosing object. Combined object descriptors proceed left-to-right from the smaller to the larger:

```
first field of last card of this background
```

This syntax lets you refer directly to any object within the current stack—you don't have to go to the card containing a particular field to get the field's contents or put something into it. For example, if the current card were the first in the stack, you could still execute the following command:

```
put the selection into field "undoHolder" of last card
```

You cannot refer to an object within another stack. You have to go to the stack before you can address its objects directly.

You can further combine field descriptors with chunk expressions, which are described in Chapter 6, "Values."

Referring to menus and menu items

HyperCard menus and menu items have names and numbers, but they don't have ID numbers.

Menu and menu item names

Menus and menu items all have names. The name of a menu or menu item is one of its properties. You assign a name for a menu when the menu is created with the `create` command. You assign a name to a menu item when you put the menu item into a menu with the `put` command.

Menu and menu item names can include any characters, including spaces. When you use a menu or menu item name in a statement, put quotation marks around the name to ensure that HyperCard recognizes it as a literal and doesn't look for a variable by that name, for example, `menuItem "User Preferences"`.

When you refer to menus by name you must precede the menu name with the word `menu`, for example, `menu "Notes"`. When you refer to existing menu items, you precede the name of the menu item with the word `menuItem`, for example, `menuItem "Power Tools"`. You must also specify which menu the menu item is in, for example, `menuItem "Power Tools" of menu "Utilities"`. You can use either `of` or `in`. There is an exception to these rules: you do not have to precede the name of a menu item with `menuItem` or specify the menu that a menu item is in when referring to a menu item in a `doMenu` statement:

```
doMenu "Power Tools"
```

Menu and menu item numbers

Menus and menu items always have numbers by which you can refer to them. The menu number refers to the menu's position within the menu bar. Menus are ordered from left to right in the HyperCard menu bar. The menu item number refers to the menu item's position within its enclosing menu. Menu items are ordered from top to bottom.

You can get the name of a menu item by referring to it with its number. For example, the statement

```
menuItem 3 of menu "Utilities"
```

puts the name of the third menu item in the Utilities menu into the Message box.

You can also use ordinals when referring to menus and menu items. For example, the statement

```
the third menuItem in the fifth menu
```

puts the name of the third menu item in the fifth menu into the Message box.

You cannot get the number of a menu or menu item by referring to it by name. For example, the statement

```
the number of menuItem "Card Info..." of menu "Objects"
```

would result in an error dialog box.

The dashed line used in menus to visually separate menu items that have different functions is also counted as a menu item in the number of menu items. For example, if you have a menu with a dashed line separating the second and third menu choices, the dashed line is referred to as `menuItem 3`. You always refer to the dashed line by a menu item number because it has no name property. To put a dashed line in an existing menu use the `put` command syntax as follows:

`put "--" after menuItem` *itemName* `of menu` *menuName*

To get a return delimited list of all of the menus in the current HyperCard menu bar, use the function `the menus`. To get a return delimited list of all the menu items in a menu, use the menu name, number, or ordinal: `menu "Power Tools"`, `menu 6`, or `the sixth menu`.

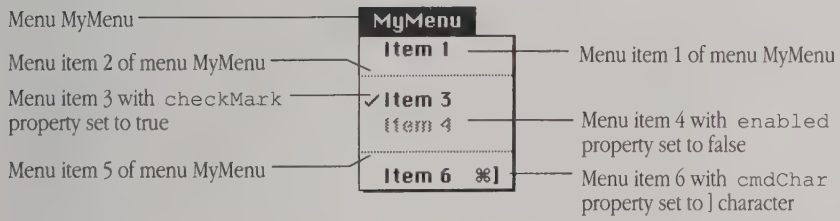
To determine how many menus are in the current HyperCard menu bar, use the function `number of menus`. To determine how many menu items are in a specified menu, use the function `number of menuItems in menu` *menuName*. To determine or change the name of a menu or menu item, use the `name` property.

You can use menu commands, functions, and properties with the HyperCard built-in menus with the exceptions noted in the descriptions of the menu commands, functions, and properties.

The following statements use some of the menu commands and properties to create the simple example of a custom (user-defined) menu shown in Figure 5-5.

```
create menu MyMenu
put "Item 1" into menu "MyMenu"
put "--" after menuItem "Item 1" of menu "MyMenu"
put "Item 3" after the second menuItem of menu "MyMenu"
put "Item 4" after menuItem "Item 3" of menu "MyMenu"
put "--" after menuItem "Item 4" of menu "MyMenu"
put "Item 6" after the fifth menuItem of menu "MyMenu"
set the enabled of menuItem "Item 4" of menu "MyMenu" to false
-- or disable menuItem 4 of menu "MyMenu"
set the cmdChar of menuItem "Item 6" of menu "MyMenu" to "]"
set the checkMark of menuItem "Item 3" of menu "MyMenu" to true
```

■ **Figure 5-5** A custom menu


Menu MyMenu
MyMenu
Item 1
Menu item 1 of menu MyMenu
Menu item 2 of menu MyMenu
Menu item 3 with checkMark property set to true
✓Item 3
Item 4
Menu item 4 with enabled property set to false
Menu item 5 of menu MyMenu
Item 6 ⌘]
Menu item 6 with cmdChar property set to] character

Referring to windows

Windows are referred to by name. Windows do not have an ID property or number property.

Card windows are referred to as `card window` (`set the location of card window to 45,65`). `Card window` always refers to the card window of the current stack. You can't use the `card window` syntax to refer to the card window of another stack unless you go to that stack first. You can, however, refer to another visible or hidden stack's window with the syntax `window "`*stackName*`"` (`set the location of window "Dizzie" to "45,65"`).

You refer to the Message Watcher window as `Message Watcher` (`show Message Watcher`) or `window "Message Watcher"` (`set the loc of window "Message Watcher" to "65,80"`).

You refer to the Variable Watcher window as `Variable Watcher` (`set the loc of Variable Watcher to "64,124"`) or `window "Variable Watcher"` (`set the hBarLoc of window "Variable Watcher" to "65,80"`).

The Tools palette and Patterns palette are referred to in HyperTalk as `tool window` and `pattern window`, or `window "tools"` and `window "patterns"`, respectively.

The Message box is referred to as `message window`, or `message box`. `Message` can be abbreviated `msg` in any of these forms.

The Scroll window is referred to as `scroll window` or `window scroll`.

In addition to the HyperCard built-in windows, there are also external windows you create with the `picture` command. They are referred to by the name you give to the window at the time of its creation. For example,

```
set the rect of window "MyBestPicture" to "60,90,300,300"
```

The function `windows()` results in a list of all the windows that are currently available to HyperCard. The list could include the Message box, Scroll window, Message Watcher window, Variable Watcher window, FatBits window, Patterns palette, Tools palette, the windows of the Home stack and any currently open stacks, and any user-defined external windows.

Chapter summary

Here is a summary of the material covered in this chapter:

- You refer to a HyperCard object using an object descriptor—its generic name and its specific designation.
- Cards, backgrounds, buttons, and fields always have unique ID numbers that never change, they always have object numbers that may change, and they may optionally be given names.
- You can use special ordinals—`middle`, `last`, and `any`—to refer to objects by their position within their enclosing object.
- You can refer to the current card, background, or stack with `this`. You can refer to the card or background preceding the current one with `previous`, and to the one following the current one with `next`. You can refer to the card that was current prior to the current one with `recent`.
- The term `me`, in a script, refers to the object containing the script.
- The only unambiguous object descriptor for a stack is the word `stack` followed by the stack's filename within quotation marks.
- You can combine object descriptors to refer directly to any object in the current stack.
- Menus and menu items can be referred to by their name or their number.
- Built-in HyperCard windows are referred to by names that are defined in the HyperTalk vocabulary. External windows you create with the `picture` command are referred to by the name you give it when you create it.

Chapter 6 **Values**

This chapter describes the elements of HyperTalk that contain values. **Values** are the information on which HyperTalk operates. A HyperTalk **expression** is a description of how to get a value.

The **sources of values** in HyperTalk are

- constants
- literals
- functions
- properties
- numbers
- containers

These sources of values are the most basic expressions.

HyperCard does not have data types: values are stored simply as strings of characters. (For mathematical operations, numbers are represented internally in a more efficient format; see "Standard Apple Numerics Environment" in this chapter.)

Constants

A **constant** is a named value that never changes. It's different from a variable in that you can't change it, and it's different from a literal in that its value is not always the string of characters making up the name. For example, the constant `empty` is the same as the null string (the literal `""`), and the constants `space` is the same as the literal `" "`. All HyperTalk constants are described in Appendix B.

Literals

A **literal** is a text string whose value *is* the string, exactly as it appears. Literals are denoted by double quotation marks at both ends of the string. (You must use the straight double quotation mark, not the printer's double quotation marks typed with the Option–left bracket and Option–Shift–left bracket keys.) Any character except double quotation mark, return, or "soft" return (generated by pressing Option-Return) can be part of a literal string. A literal can be of any length. For example, `"This is a literal string"` is a literal.

- *Unquoted literals are not supported:* Do not use unquoted literals in HyperTalk. The value of an unquoted literal is the literal of itself—as though you had entered `put "fred" into fred`. Always put double quotation marks around a word you want HyperCard to take as a literal.

Functions

A **function** is a named value that HyperCard calculates when the statement in which the function is used executes. The value of a function varies according to conditions of the system or according to the value of parameters you pass to the function when you use it.

For example, the built-in function named `the time` returns the current time in place of itself in a HyperTalk statement:

```
put the time into the message box
```

If the current time were 5:12 P.M., the above example would put `5:12 PM` into the Message box.

You can also define your own functions in scripts using the function handler structure described in Chapter 9.

All built-in HyperTalk functions are described in Chapter 11.

Properties

A **property** is a named value representing one of the defining characteristics of an element of the HyperCard environment. Different types of objects and other elements have different properties, according to their purpose. For example, fields share a set of properties, many of which are different from the set shared by buttons.

You get the value of most properties by using the property name as a function in a script or in the Message box. For example, the following statement retrieves the `location` property (two integers separated by a comma) of button 1, and it puts the value into the Message box:

```
put the location of button 1 into msg
```

This next example returns a value of either `true` or `false` for the `enabled` property of the menu Clues.

```
put the enabled of menu "Clues"
```

You can also change most properties with the `set` command. All HyperCard properties are described in Chapter 12.

Numbers

A **number** in HyperCard is a character string consisting of any combination of the numerals 0 through 9, representing a decimal value. A number can include one period (.) representing the decimal point, but it can have no other punctuation nor a space character. A number can be preceded by a hyphen or minus sign to represent a negative value (HyperCard doesn't recognize a plus sign as part of a number). Numbers that consist only of numerals are integers. Numbers that include a period are real, and, when used with mathematical operators, are manipulated with floating-point operations.

Standard Apple Numerics Environment

HyperCard performs mathematical operations with Standard Apple Numerics Environment (SANE) routines, but you don't have to worry about how to represent the values. You always enter numbers into HyperCard containers as numeric strings.

When performing a mathematical operation, HyperCard automatically converts the strings representing the numbers to SANE numeric values. If you put the result of the operation into a variable, it's stored as a SANE numeric value; if you put it into a field or the Message box, HyperCard automatically converts it back to a string with a precision of up to 19 decimal places. The same conversion takes place if you put the variable into a field or the Message box at a later time, or if you use it in a way that implies a string (`character 2 of varName`). So although SANE values are used internally for handling numbers with speed and precision, you can always think of HyperTalk numbers as strings.

Precision

The precision of the decimal string, resulting from putting a SANE numeric value into a field or the Message box, is controlled by the `numberFormat` global property. (See Chapter 12 for a detailed description.) For example, the command

```
set numberFormat to 0.00
```

would result in a string with at least one digit to the left of the decimal point and exactly two digits to the right of the decimal point.

The `numberFormat` property does not affect the precision with which mathematical operations are executed, only the precision with which the results are displayed. When you put a number into a field or the Message box to display it, however, HyperCard converts it to a decimal string. So any extra precision it may have had (beyond the `numberFormat` specification in effect at the time) is lost.

Number handling

The following example shows how number handling works. These three HyperTalk statements put the constant `pi` into a variable, set the `numberFormat` property, and put the value of the variable into the Message box, respectively:

```
put pi into joe
set numberFormat to 0.00
put joe into msg
```

The result shown in the Message box is 3.14159265358979323846. In this case, `pi` is entered into the variable `joe` as a string, and it remains a string, so `numberFormat` has no effect. If, however, you perform a mathematical operation on the variable, HyperCard converts it to a SANE numeric value:

```
put pi into joe -- joe contains a string
add 0 to joe -- mathematical operation makes it a number
set numberFormat to 0.00 -- affects the format of joe
put joe into msg
```

The result shown in the Message box is `3.14`. In this case, `numberFormat` takes effect when `joe` is converted from a SANE numeric value to a string as it's put into the Message box. The example statements for number handling work only when placed inside a handler. If entered one at a time in the Message box, the result is in the default format, because HyperCard resets `numberFormat` to its default value during idle time.

Containers

A **container** in HyperCard is a place where you can store a value. Containers include fields, variables, the current selection, and the Message box. Containers other than fields can store values of any length, including zero length. Containers other than the Message box can have more than one line in them; each line ends with a return character (which can be the only character in the line).

Fields

A **field** is a HyperCard object for holding and displaying editable text. Fields are unique objects because they are also containers—a field's value is the text string it contains. Fields can also act as expressions, for example, `put field 1 into it`, puts the value of the expression, `field 1`, into the variable `it`. Variables are described in the next section and expressions are described in Chapter 7.

You can refer to fields directly by name, number, or ID number. (See Chapter 5, "Referring to Objects, Menus, and Windows," for more about how to refer to fields.)

Fields belong to cards or to backgrounds; the text held by a field, however, always remains with the card, even if the field belongs to the background. A field can contain up to 30,000 characters, including spaces, return characters, and other invisible characters. If you put more than that many characters into a field, the extras are ignored.

You can search through text with the `find` command unless the `dontSearch` property of the field is true. You can edit it using the I-beam pointer of the Browse tool when the field isn't locked.

- *About Paint text:* You can also put text onto cards and backgrounds as **Paint text**—pictures that look like characters. Paint text can't be edited once it has been fixed onto the card or background (although you can paint over it or erase it as you can any part of a picture). See the *HyperCard Reference* for more information on Paint text.

Variables

A **variable** is a named container that has no visible representation other than its name. Its value is a character string of any length. The variable name is a HyperTalk **identifier.** An identifier can be of any length, it always begins with an alphabetic character, and it can contain any alphanumeric character plus the underscore character (_).

You assign a value to a variable with the `put` command. You cannot read from a nonexistent variable—you must create it by putting something into it before you use it. The constant `empty`, the null string, counts as something you can put into a variable. This example puts a numeric value `12` into a the variable `fudge`, adds 5 to that variable, and then puts the result, `17`, into the Message box. Enter each line separately in the Message box.

```
put 12 into fudge
add 5 to fudge
put fudge into msg
```

HyperCard assumes that an unquoted word used in an expression is a variable when it can't interpret the word as some other source of value (the string is not a function, constant, property, or other container name). If you haven't put a value into a variable by that name, HyperCard treats it as an unquoted literal.

Scope of variables

HyperCard has both local and global variables. A **local variable** is valid only during the current invocation of the currently executing handler. You don't need to declare a local variable before you use it—just put something into it. A **global variable** is valid for all handlers. You must declare a variable as global by using the `global` keyword in each handler before you use the variable:

```
global useMeEverywhere,useMeToo
```

HyperTalk assumes a variable to be local unless you specifically use the `global` keyword.

For more details on the `global` keyword, see Chapter 9.

Parameter variables

You create parameter variables when you put their names after the message name in a handler:

```
on messageName firstParam,secondParam
```

When the handler is called, these variables are assigned the values, if any, of the items in a comma-separated list of expressions following the message name in the calling statement. Parameter variables are local to their handler. See Chapter 4, "Handling Messages," for an explanation of parameter passing.

The variable `It`

The local variable named `It` is the destination of the commands `get`, `ask`, `answer`, and `read`. For example, `get the name of field 1` puts the value of that background field's name into `It`. `Convert` puts its results into `It` if another destination isn't specified.

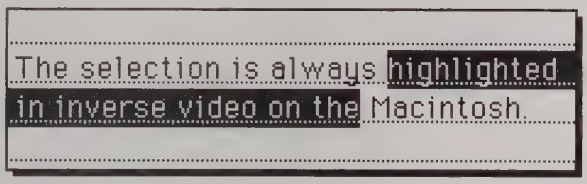
For information on these commands, see Chapter 10.

The selection

The **selection** is a container that holds the currently selected area of text. You can put values into, before, or after the selection or put the selection (or any chunk of the selection) into another container. Figure 6-1 shows an example that puts a string into the selection to replace the highlighted text.

■ **Figure 6-1** Manipulating the selection

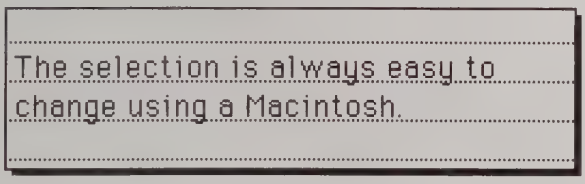
Starting with this selection ...



... this HyperTalk command ...

put "easy to change using a" into the selection

... produces this result.



If the phrase `I'm the selected text` is selected, and your handler issues the statement

```
put the selection into the Message box
```

then `I'm the selected text` appears in the Message box. (Both instances of the word `the` in the example are optional.)

- *Found text isn't selected:* Text found by the `find` command is indicated by a box around it—it is not placed into `selection`. To get information about text found with the `find` command, use the functions `the foundText`, `the foundChunk`, `the foundLine`, and `the foundField`, which are described in Chapter 11, "Functions."

You must select some text with the mouse or the `click` or `drag` command before you can manipulate the selection container.

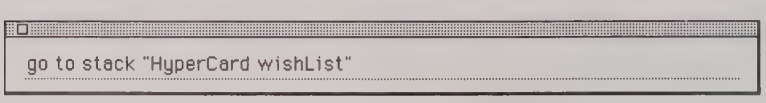
You can also get information about a chunk of text or a line in a field that has been clicked with the `clickChunk` and `clickLine` functions described in Chapter 11.

The Message box

The **Message box** is a special container. Typically, you use the Message box to send a HyperTalk message directly to an object or to HyperCard. The Message box is a single-line container, as shown in Figure 6-2. If you put more than one line from a multiple-line container into the Message box (`put card field 2 into msg`), only the first line is copied into the Message box.

- *Referring to the Message box:* There are several forms you can use when referring to the Message box. The forms are `message box`, `message`, and `msg`.

■ **Figure 6-2** The Message box



The Message box is the default destination for the `put` command. You can put a value directly into the Message box without specifying the Message box by using one of these forms of the `put` command.

`put` *property* `[of` *element*`]`
`put` *container*
`put` *function*

Property is an expression that yields any HyperCard property; *element* yields the descriptor of a HyperCard element (an object, menu, menu item, or window) *container* yields a container; and *function* yields any HyperCard function.

If you put something into the Message box when it's hidden, HyperCard shows it automatically. You can toggle the Message box between being hidden or shown by pressing Command-M.

The Message box can be specified by just the word `message` or its abbreviation `msg`. Optionally, you can follow either of those with either `box` or `window`, and you can precede either with the word `the`.

See the description of the `put` command in Chapter 10, "Commands," for more information about the values you can put into the Message box.

Chapter summary

Here is a summary of the material covered in this chapter:

- The most basic expressions in HyperTalk are constants, literals, functions, properties, numbers, and containers.
- HyperTalk values can always be treated as strings of characters.
- Containers—fields, variables, the selection, and the Message box—are places to store values.

Chapter 7 **Expressions**

This chapter describes the expressions you use to refer to values. An **expression** is a description of how to get a value. It may be as simple as a single source of a value, or it can be a complex expression built with operators.

It also describes HyperTalk **operators,** the elements of the language that you use in expressions to manipulate and calculate values, described in Chapter 6.

Complex expressions

You can build complex expressions using values and operators. As a complex expression is evaluated, the values of its basic components are manipulated to derive a final value in place of the entire expression. (The original values are not changed in the process.) Complex expressions are evaluated according to rules of precedence, and restrictions apply to the values that can be used, depending on their operators.

- *Chunk expressions are different:* Chunk expressions are a different type of expression: they designate pieces of the strings representing values. Chunk expressions are described later in this chapter.

Factors

A **factor** is a single element of value in an expression. The following constructs are factors:

- a simple source of value
- an expression enclosed in parentheses
- an element (which must evaluate to a number) with a minus sign in front of it
- an expression (which must evaluate to `true` or `false`) with the word `not` in front of it

An expression can be a single source of value, or it can be any two expressions with an operator between them.

The difference between a factor and an expression is important to the syntax of HyperTalk commands and functions. Where a built-in HyperTalk command parameter permits an expression, you can specify as complex an expression as you wish. HyperCard derives the final value before passing the parameter to the command. For example, the `add` command accepts a complex expression as its first parameter:

```
add 46+12*monthlyRate to total
```

In contrast, when a built-in HyperTalk function requires a factor, HyperCard takes the value of the first factor as the parameter to pass to the function. For example, the `sqrt` function takes the first factor following its name as its parameter. This is illustrated by the following expression, which you can type into the Message box or use in a statement:

```
the sqrt of 4+12
```

In the example, the `sqrt` function takes the factor `4` as its parameter, rather than the value of the expression `4+12`. So the entire expression evaluates to 14, rather than 4, which would be the value if `sqrt` accepted an entire expression. (To specify the entire expression `4+12` as the parameter, you can enclose it in parentheses, which turns it into a single factor.)

- *Two hyphens always indicate a comment:* You can put a hyphen in front of a factor to create another factor, and you can put another hyphen in front of that and still have a factor. However, two hyphens in sequence indicate a comment, so you must separate the hyphens with a space or enclose the inner factor in parentheses for HyperCard to recognize the construct as a factor.

HyperTalk built-in commands and functions are described in Chapters 10 and 11, respectively.

HyperTalk operators

Operators are used in complex expressions to derive values from other values. Operators fall into several categories:

- Arithmetic operators work on numbers and result in numbers.
- Comparison operators work on numbers, text, and Boolean values (`true` or `false`) and result in Boolean values.
- Logical operators work on Boolean values and result in Boolean values.
- Text operators manipulate text strings and result in text strings.

Table 7-1 is a list of all the operators in HyperTalk.

■ **Table 7-1** HyperTalk operators

Operator	Description
`( )`	Grouping: Expressions within the innermost pair of parentheses are evaluated first. Parentheses don't force a new level of evaluation; they change the sequence in which the current level of evaluation proceeds.
`-`	Minus: Arithmetic operator that makes negative the number to its right, or, if it is between two numbers, subtracts the one on the right from the one on the left.
`+`	Plus: Arithmetic operator that adds two numbers it appears between.
`*`	Multiply: Arithmetic operator that multiplies two numbers it appears between.
`/`	Divide: Arithmetic operator that divides the number to its left by the number to its right.
`div`	Divide and truncate: Arithmetic operator that divides a number to its left by a number to its right, ignoring any remainder, resulting in just the whole part.
`mod`	Modulo: Arithmetic operator that divides the number to its left by the number to its right, ignoring the whole part, resulting in just the remainder.
`^`	Exponent: Arithmetic operator that raises the number to its left to the power of the number to its right.
`not`	NOT: Logical operator that results in `true` if the expression on its right is `false`, and `false` if the expression on its right is `true`.
`and`	AND: Logical operator that results in `true` if both the expression to its left and the expression to its right are `true`.
`or`	OR: Logical operator that results in `true` if either the expression to its left or the expression to its right is `true`.
`=`	Equal: Comparison operator that results in `true` if the expression to its left and the expression to its right have the same value. The expressions can be arithmetic, text string, or logical.
`is`	Is: Same as `=`.
`is a` or `is an`	Is a, is an: Same as =. Comparison operator that tests for types. Types include: `number`, `integer`, `point`, `rect`, `date`, and `logical`.

[continued]

■ **Table 7-1** HyperTalk operators [continued]

Operator	Description
<>	Not equal: Comparison operator that results in `true` if the expression to its left and the expression to its right have different values. The expressions can be arithmetic, text, or logical.
≠	Not equal: Same as <>. The ≠ character is obtained on the Macintosh keyboard by pressing Option-equal sign (=).
`is not`	Is not: Same as <>.
`is not a` or `is not an`	Is not a, is not an: Same as <>. Comparison operator that tests for types. Types include: `number`, `integer`, `point`, `rect`, `date`, and `logical`.
<	Less than: Comparison operator that results in `true` if the expression to its left has less value than the one to its right. The expressions can be both arithmetic or both text.
>	Greater than: Comparison operator that results in `true` if the expression to its left has greater value than the one to its right. The expressions can be both arithmetic or both text.
<=	Less than or equal to: Comparison operator that results in `true` if the expression to its left has less value than the one to its right or the same value. The expressions can be both arithmetic or both text.
≤	Less than or equal to: Same as <=. The ≤ character is obtained on the Macintosh keyboard by pressing Option-comma (,).
>=	Greater than or equal to: Comparison operator that results in `true` if the expression to its left has greater value than the one to its right or the same value. The expressions can be both arithmetic or both text.
≥	Greater than or equal to: Same as >=. The ≥ character is obtained on the Macintosh keyboard by pressing Option-period (.).
`contains`	Contains: Comparison operator that results in `true` if the text string yielded by the expression on its right is found in the text string yielded by the expression on its left.
`is in`	Is in: Converse of `contains`; comparison operator that results in `true` if the text string yielded by the expression on its left is found in the text string yielded by the expression on its right.

[continued]

■ **Table 7-1** HyperTalk operators [continued]

Operator	Description
`is not in`	Is not in: Opposite of `is in`; comparison operator that results in `true` if the text string yielded by the expression on its left is not found in the text string yielded by the expression on its right.
`there is a` or `there is an`	There is a, there is an: Comparison operator that results in `true` if the item exists. Items include the descriptor of a window, menu, menu item, file, button, field, card, background, and stack.
`there is not a` or `there is not an`	There is not a, there is not an: Opposite of `there is a`; comparison operator that results in `true` if the specified item does not exist. Items include the descriptor of a window, menu, menu item, file, button, field, card, background, and stack.
`within`	Within: The `within` operator tests whether or not a point lies inside a rectangle; it results in a Boolean value: `true` or `false`.
`&`	Concatenate: Text string operator that joins the text string yielded by the expression on its left to the text string yielded by the expression on its right.
`&&`	Concatenate with space: Text string operator that joins the text string yielded by the expression on its left to the text string yielded by the expression on its right, with a space between them.

Operator precedence

Parentheses alter the order of expression evaluation. Different operators have different orders of precedence that determine how things get evaluated. The order in which HyperCard performs operations is shown in Table 7-2.

■ **Table 7-2** Operator precedence

Order	Operators	Type of operator
1	`( )`	Grouping
2	`-`	Minus sign for numbers
	`not`	Logical negation for Boolean values
	`there is a`	Comparison for HyperCard items
	`there is an`	Comparison for HyperCard items
	`there is not a`	Comparison for HyperCard items
	`there is not an`	Comparison for HyperCard items
	`within`	Boolean test for point within rectangle
3	`^`	Exponentiation for numbers
4	`* / div mod`	Multiplication and division for numbers
5	`+ -`	Addition and subtraction for numbers
6	`& &&`	Concatenation of text
7	`> < <= >= ≤ ≥`	Comparison for numbers or text
	`is in contains`	Comparison for text
	`is not in`	Comparison for text
	`is a`	Comparison for types
	`is an`	Comparison for types
	`is not a`	Comparison for types
	`is not an`	Comparison for types
8	`= is is not <> ≠`	Comparison for numbers or text
9	`and`	Logical for Boolean values
10	`or`	Logical for Boolean values

Operators of equal precedence are evaluated left to right, except for exponentiation, which goes right to left. For example, `2^3^4` means "3 raised to the fourth power, then 2 raised to that power," whereas `1-2-3` means "2 subtracted from 1, then 3 subtracted from that." If you use parentheses, HyperCard evaluates the parenthetical expression first.

Operators and expression type

The operator you use must match the values you're using it with: `"tom" + "cat"` would cause an error, because numeric values are required for addition. On the other hand, `tom + cat` would be acceptable if `tom` and `cat` were names of containers with numbers in them, and `"tom" & "cat"` would be acceptable because the `&` operator works on text strings (the result of this operation would be the text string `tomcat`). Text operators work on any value, because any value in HyperTalk can be treated as a text string; they always yield text strings.

Because numeric values are automatically converted to strings when necessary (see "Numbers" in Chapter 6), they can be manipulated by both text operators and arithmetic operators. For example, `5 & 78` yields `578`, and `5 + 78` yields `83`.

Comparison operators try to treat both of their operands as numbers; if they can't both be regarded as numbers, HyperCard treats them as text and does a lexical comparison. A lexical comparison uses the order of the ASCII table (See Appendix D). For letters, it's the same as alphabetical order; for numerals it's 0–9, but it's different from a numerical comparison because a lexical comparison looks at just one character at a time, rather than the number as a whole. For example, `9 < 10` results in `true`, because `9` is less than `10` arithmetically. But `"9x" < "10x"` results in `false`, because the operands are evaluated lexically and `9` is greater than `1`.

Chunk expressions

You use a chunk expression to specify a particular piece—a **chunk**—of the value of any source of value: constant, literal, function, property, number, or container. Chunk expressions can specify any character, word, item, or line in the source.

Syntax of chunk expressions

The form of a chunk expression designates the smallest part of the chunk first, then specifies each larger, enclosing part. You separate each part of the expression with the preposition `of` or its synonym `in`. For example, the expression

```
first character of second word of third line of field 1
```

specifies a single character in the field.

You modify the specification of the kind of chunk—character, word, item, or line— with the number of the particular one you want. The number can be an ordinal constant preceding the kind (`tenth word`) or an integer following the kind (`line 2`). You can also use a numeric constant in place of the integer (`line two`), or any numeric expression that resolves to an integer.

You can use the special ordinals `middle`, `last`, and `any` to specify a chunk within its enclosing part. HyperCard resolves a special ordinal to a number using the total number of chunks of the specified type within its enclosing part: `middle` resolves to one more than half the total (rounded down to the nearest integer), `last` resolves to the total, and `any` resolves to a random number between 1 and the total. For example,

```
put "Joe" into any word of line 2 of field 1
```

replaces a random word in the line with `Joe`.

It isn't necessary to specify the enclosing parts of the source in strict hierarchical order. You can designate any smaller part within any larger part:

```
character 35 of field 1
```

And, although you must go left-to-right from smaller to larger, you don't have to specify any smaller part than you want:

```
third item of It
```

Characters

Characters are designated by the chunk name `character` (or `char`). Spaces count as characters in any part of a source except words. (Words are delimited by spaces.) Commas count as characters except in items. (Items are delimited by commas.) Return characters count as characters in whole sources and items. (A return character delimits the last word on the line as well as the line itself.)

For example, if field 6 contains the phrase

```
It was the turtle, not I, who spilled the beans.
```

the chunk expression

```
character 25 of field 6
```

yields a comma (the one after `not I`).

Words

Words are composed of any characters, including punctuation, delimited by spaces, and are designated by the chunk name `word`:

```
word 2 of "Where's my cubicle?"
```

yields `my`.

Items

Items are composed of any characters, including punctuation, delimited by commas, and are designated by the chunk name `item`:

```
item three of "cat's, rat's, bat's, gnat's"
```

yields `" bat's"` (including the space character in front).

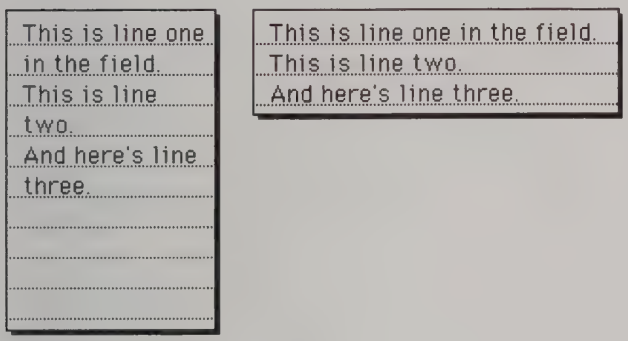
Lines

Lines are composed of any characters, including punctuation, delimited by return characters, and are designated by the chunk name `line`.

The chunk name `line` denotes text between the beginning of a container and the first return character, between two return characters, or between the last return character and the end of the container.

It doesn't matter how many display lines it takes to display one container line. For example, a single line in a field might occupy several lines on the display if the text wraps around (which it does if the field isn't wide enough to accommodate the whole line).

■ **Figure 7-1** Lines in a field



Ranges

The preposition `to` in a chunk expression specifies a range of a chunk within the larger chunk:

```
word 1 to 5 of line 2 of field "fred"
```

The numbers given in a range are inclusive. For example,

```
char 2 to 5 of "Hedgehog"
```

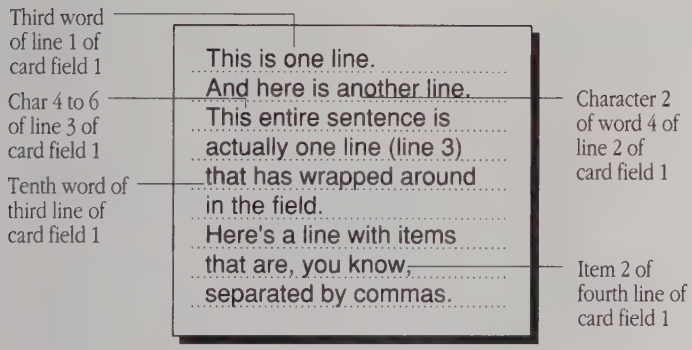
yields `edge`.

You specify the range with integers (or with constants or numeric expressions that resolve to integers) following the chunk name, rather than with ordinal numbers preceding the chunk name. That is, you must say `char 1 to 3 of "george"`; you can't say `first to third char of "george"`.

When the first number in a range is greater than the second, you get an empty string. For example, `char 5 to 3 of "Motorcycle"` yields `""` or `empty`.

Figure 7-2 shows some chunk expressions, labeled in various valid forms of chunk expression syntax, in a hypothetical `card field 1`.

■ **Figure 7-2** Chunk expressions


Third word of line 1 of card field 1
Char 4 to 6 of line 3 of card field 1
Tenth word of third line of card field 1
This is one line.
And here is another line.
This entire sentence is actually one line (line 3) that has wrapped around in the field.
Here's a line with items that are, you know, separated by commas.
Character 2 of word 4 of line 2 of card field 1
Item 2 of fourth line of card field 1

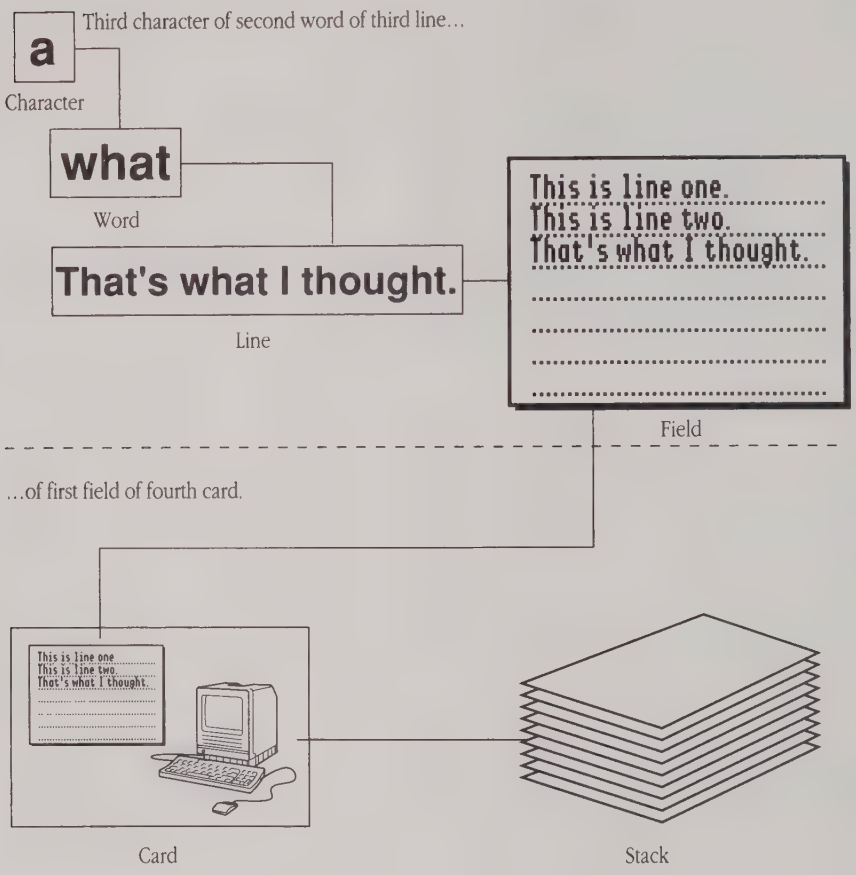
Chunks and containers

Combining a chunk expression with the object descriptor of a field lets you refer directly to any piece of text down to a single character within the current stack:

```
put char 2 of line 2 of field 1 of last card
```

- *You can't specify chunks in another stack:* You can't combine a stack name with a chunk expression; you must go to the stack first.

■ **Figure 7-3** Combining chunks and objects



Chunks as destinations as well as sources

Chunk expressions can be used to specify a part of the value in a container wherever a container name is used. So, the chunk can specify the destination of a value—where you're putting it—as well as the source of a value—where you're getting it. For example,

```
put "Mr Steve" into word 3 of field 1
```

replaces only the third word in the field with the value `Mr Steve`, leaving the rest of the field's former contents intact.

Nonexistent chunks

If you specify chunks that don't exist as sources of values, you get nothing. For example,

```
put char 5 of "hey" into msg
```

puts `empty` (or nothing) into the Message box, because the word *hey* contains only three characters.

If you specify a nonexistent chunk as the destination of a `put` command, the outcome depends on the kind of chunk. If you put a value into a character or a word that doesn't exist in a container, you put just the value. That is, if field 1 is empty, the statement

```
put "hey" into word 5 of field 1
```

puts `hey` (with no characters before it) into background field 1.

If you put a value into a nonexistent line, however, HyperCard puts in a return character, and if you put a value into a nonexistent item, HyperCard puts in a comma. (In both cases, you put a null chunk delimited by its particular delimiting character.) For example, if field 1 is empty, the statement

```
put "hey" into line 5 of field 1
```

puts four return characters (four null lines) followed by `hey` into background field 1. Similarly,

```
put "hey" into item 5 of field 2
```

puts four commas (four null items) followed by `hey` into the first line of background field 2.

Chapter summary

Here is a summary of the material covered in this chapter:

- Complex expressions are built with values and operators.
- Operators are used to manipulate and calculate values.
- Chunk expressions can specify any chunk—character, word, item, or line—either in a source of value or as the destination of a `put` command.

Chapter 8 **System Messages**

This chapter describes the messages HyperCard sends in response to events, such as mouse clicks, that you initiate in its environment.

Most system messages are sent by HyperCard to the current card, but those having to do with a specific button or field are sent to that object. The receiving object has the first chance to respond to the message before it goes on to the next encompassing object, as described in Chapter 4, "Handling Messages." The receiving object can respond to the system message with a handler that begins

```
on messageName
```

where *messageName* is one of the system messages in the lists in this chapter.

Messages and commands

Most system messages are informational—they cause no action if passed all the way to HyperCard, although they may be a result of a HyperTalk command executing. For example, HyperCard sends `deleteButton` to a button while it is executing either a Cut Button or Clear Button menu command. The `deleteButton` message is a result of a command, not the command itself. Consequently, you can't prevent the deletion of buttons by intercepting the `deleteButton` message with a handler named `deleteButton`. All system messages that are a result of a command can be intercepted, but the intercepting handler will have no affect on the action of the command that sends that message.

Other system messages, however, are commands if passed to HyperCard. For example, all menu commands are passed to HyperCard as parameters of the `doMenu` message. (So you can prevent deletion of buttons by intercepting `doMenu`. But see the section "Redefining Commands" at the beginning of Chapter 10 before trying it.) All system messages that are HyperTalk commands are noted as such in this chapter and are also listed in Chapter 10. If a message that reaches HyperCard is neither a system message nor a command, HyperCard displays a "Can't understand" error dialog box.

Although system messages are usually sent by HyperCard, they can be sent by other objects as well. For example, a handler could invoke a `mouseUp` handler in another object by executing a statement such as

```
send "mouseUp" to button 1 of card 1
```

The tables in this chapter correspond to the type of object to which the listed system messages are sent initially. If that object has no handler with a name matching the system message, it passes the message on to succeeding objects in the hierarchy. So, for example, a card can have a handler for a message sent initially to a button.

Messages sent to a button

The only messages that are sent initially to buttons are those having to do with a specific button. They are of two types: those announcing the button's creation or deletion, and mouse messages.

All of the mouse messages that can be sent to buttons can also be sent to fields. When buttons and fields are layered on top of each other, mouse messages are sent only to the closest one. Background buttons and fields can never overlay those belonging to the card. Both background buttons and card buttons precede the card in the message-passing hierarchy even though the background itself comes after the card.

Table 8-1 shows the system messages HyperCard sends initially to buttons.

■ **Table 8-1** Messages sent to a button

Message	Meaning

`newButton`	Sent to a button as soon as it has been created. Because the new button can have no script with which to respond to this message (unless it was created by pasting), the message passes to objects lower in the hierarchy which can respond with handlers such as
	`on newButton`
	`  set autoHilite of the target to true`
	`end newButton`

`deleteButton`	Sent to a button that is being deleted, just before it disappears.
`mouseDown`	Sent to a button when the mouse button is pressed down while the pointer is inside its rectangle. (This message may also be sent to a field or card; see Tables 8-2 and 8-3.)
`mouseStillDown`	Sent to a button repeatedly while the mouse button is held down and the pointer is inside its rectangle. (This message may also be sent to a field or card; see Tables 8-2 and 8-3.)
`mouseUp`	Sent to a button when the mouse button is released while the pointer is inside its rectangle. The pointer must be in the same button rectangle it was in when the mouse button was pressed down for the message to be sent. (This message may also be sent to a field or card; see Tables 8-2 and 8-3.)
`mouseEnter`	Sent to a button as soon as the pointer is moved within its rectangle. (This message may also be sent to a field; see Table 8-2.)

[continued]

■ **Table 8-1** Messages sent to a button [continued]

Message	Meaning
`mouseWithin`	Sent to a button repeatedly while the pointer is inside its rectangle. (This message may also be sent to a field; see Table 8-2.)
`mouseLeave`	Sent to a button as soon as the pointer is moved outside its rectangle. (This message may also be sent to a field; see Table 8-2.)

Messages sent to a field

The only messages that are sent initially to fields are those having to do with a specific field. They are of three types: those announcing the field's creation or deletion, those announcing its opening for text entry or closing afterwards, and mouse messages.

All of the mouse messages that can be sent to fields can also be sent to buttons. When buttons and fields are layered on top of each other, mouse messages are sent only to the closest one. Background buttons and fields can never overlay those belonging to the card. Both background fields and card fields precede the card in the message-passing hierarchy even though the background itself comes after the card.

Table 8-2 shows the system messages HyperCard sends initially to fields.

■ **Table 8-2** Messages sent to a field

Message	Meaning
`newField`	Sent to a field as soon as it has been created.
`deleteField`	Sent to a field that is being deleted, just before it disappears.
`enterInField`	Sent to a field when the Enter key is pressed while there is an insertion point or selection in the field. If `enterInField` is not intercepted by a handler and the contents of the field have been changed, HyperCard sends the `closeField` message.

[continued]

■ **Table 8-2** Messages sent to a field [continued]

Message	Meaning
`returnInField`	Sent to a field when the Return key is pressed while there is an insertion point or selection in the field. In response to a `returnInField` message, HyperCard sends a `tabKey` message to the field if the following conditions are true: ■ The field's `autoTab` property (described in Chapter 12) is `true`. ■ `ReturnInField` is not intercepted by a handler. ■ The field is not a scrolling field. ■ The insertion point or selection is on the last line. Otherwise, HyperCard inserts a return character into the field. The `tabKey` message, if it's not intercepted, causes HyperCard to place the insertion point in the next field.
`openField`	Sent to an unlocked field when it is opened for text editing by clicking in the field or moving the text insertion point from the previous field with the Tab key.
`closeField`	Sent to an unlocked field when it is closed after text editing by clicking outside the field, moving the text insertion point to the next field with the Tab key, pressing the Enter key, clicking a button, going to another card, or quitting HyperCard. The message is not sent unless some text was actually changed.
`mouseDown`	Sent to a locked field when the mouse button is pressed down while the pointer is inside it. `MouseDown` is not sent to a scrolling field when the mouse is clicked in the scroll bar. You can send `mouseDown` to an unlocked field by holding down the Command key while clicking the mouse in the field. (This message may also be sent to a button or card; see Tables 8-1 and 8-3.)
`mouseStillDown`	Sent to a locked field repeatedly while the mouse button is held down and the mouse pointer is inside it. (This message may also be sent to a button or card; see Tables 8-1 and 8-3.)

[continued]

■ **Table 8-2** Messages sent to a field [continued]

Message	Meaning
`mouseUp`	Sent to a locked field when the mouse button is released while the pointer is inside it. The pointer must be in the same field it was in when the mouse button was pressed down for the message to be sent. (This message may also be sent to a button or card; see Tables 8-1 and 8-3.)
`mouseEnter`	Sent to a field as soon as the pointer is moved into it. (This message may also be sent to a button; see Table 8-1.)
`mouseWithin`	Sent to a field repeatedly while the pointer is inside it. (This message may also be sent to a button; see Table 8-1.)
`mouseLeave`	Sent to a field as soon as the pointer is moved outside it. (This message may also be sent to a button; see Table 8-1.)
`tabKey`	Sent to a field when the Tab key is pressed while the text insertion point is in the field. (This message may also be sent to the current card; see Table 8-3.)

Messages sent to the current card

System messages not sent to buttons or fields are sent initially to the current card, even when they concern the background or the stack.

Mouse messages are sent to the card only when there is no button or field, belonging to either the card or the background, under the pointer.

Table 8-3 shows the system messages HyperCard sends initially to the current card.

■ **Table 8-3** Messages sent to the current card

Message	Meaning
`newCard`	Sent to a card as soon as it has been created.
`deleteCard`	Sent to a card that is being deleted, just before it disappears.
`openCard`	Sent to a card when you go to it.
`openPalette`	Sent to the current card when a palette is opened with the `palette` command.
`openPicture`	Sent to the current card when a window is created with the `picture` command.
`close`	Sent when you close a stack window with the `close window` command and when you click the close box of a card window.
`closeCard`	Sent to a card when you leave it.
`closePalette`	Sent to the current card when a palette that was opened with the `palette` command is closed.
`closePicture`	Sent to the current card when a window that was created with the `picture` command is closed.
`mouseDown`	Sent to the current card when the mouse button is pressed down and the pointer is not in a button rectangle or field. (This message may also be sent to a button or field; see Tables 8-1 and 8-2.)
`mouseStillDown`	Sent to the current card repeatedly while the mouse button is held down. (This message may also be sent to a button or field; see Tables 8-1 and 8-2.)
`mouseDownInPicture`	Sent to the current card when the mouse button is held down in a window created with the `picture` command. See also the `picture` command in Chapter 10.
`mouseUp`	Sent to the current card when the mouse button is released. (This message may also be sent to a button or field; see Tables 8-1 and 8-2.)
`mouseUpInPicture`	Sent to the current card when the mouse button is released after being down in a window created with the `picture` command. See also the `picture` command in Chapter 10.
`startUp`	Sent to the first card displayed when HyperCard is started.

[continued]

■ **Table 8-3** Messages sent to the current card [continued]

Message	Meaning

`idle`	Sent to the current card repeatedly when nothing else is happening and the Browse tool is current.
	An `idle` handler can interfere with typing. For example, if you have an `idle` handler that puts text into a field, it can remove the insertion point from another field while the user is typing. Here is an example of such a handler:
	`on idle`
	`  put the time into card field "Time"`
	`  pass idle`
	`end idle`
	If this `on idle` handler were to execute during typing into another field (`idle` is sent during a typing pause), and if the time had changed, HyperCard would remove the insertion point from the user's field. The user would have to click in the field or press Tab to replace the insertion point after every pause, which would be annoying and tedious.

`returnKey`	Sent to the current card when the Return key is pressed unless the text insertion point is in a field. (This message is also a HyperTalk command; see Chapter 10.)
`enterKey`	Sent to the current card when the Enter key is pressed unless the text insertion point is in a field. (This message is also a HyperTalk command; see Chapter 10.)
`tabKey`	Sent to the current card when the Tab key is pressed unless the text insertion point is in a field. In that case, `tabKey` is sent initially to the field; see Table 8-2. (This message is also a HyperTalk command; see Chapter 10.)

[continued]

■ **Table 8-3** Messages sent to the current card [continued]

Message	Meaning

`arrowKey` *var*	Sent to the current card when an arrow key is pressed (see also the `textArrows` property in Chapter 12). The value passed into the parameter variable *var* can be `left`, `right`, `up`, or `down`, depending on which arrow key is pressed. The beginning of a handler for this message could read
	`on arrowKey whichKey`
	`  if whichKey = "left" then go previous card`
	`  ...`
	(This message is also a HyperTalk command; see Chapter 10.)

`functionKey` *var*	Sent to the current card when a function key on the Apple Extended Keyboard is pressed. The parameter variable *var* can range from `1` to `15`. Function keys 1 through 4 are preprogrammed for the Undo, Cut, Copy, and Paste commands, respectively. The beginning of a handler for this message could read
	`on functionKey whichKey`
	`  if whichKey < 5 then pass functionKey`
	`  else if whichKey is 5 then doMenu "New Card"`
	`  else if whichKey is 6 then choose browse tool`
	`  else if whichKey is 7 then choose button tool`
	You can override the preprogrammed functions of keys 1 through 4 in a `functionKey` message handler. (This message is also a HyperTalk command; see Chapter 10.)

`commandKeyDown` *var*	Sent to the current card when a combination of the Command key and another key is pressed. The parameter variable *var* can be any character on the keyboard. The beginning of a handler for this message could read
	`on commandKeyDown whichKey`
	`  if whichKey = j then doMenu "MyMenu"`
	`  -- more statements...`

[continued]

■ **Table 8-3** Messages sent to the current card [continued]

Message	Meaning

`controlKey` *var*	Sent to the current card when a combination of the Control key and another key is pressed. The possible values of the parameter value *var* and the keys each value corresponds to are shown in Appendix D. (See also the `controlKey` command in Chapter 10.) The beginning of a handler for this message could read
	`on controlKey whichKey`
	`  if whichKey = 16 then doMenu "Print Card"`
	`  -- additional statements...`

`keyDown` *var*	Sent to the current card when a key is pressed. The parameter variable *var* can be any character on the keyboard. The beginning of a handler for this message could read
	`on keyDown whichKey`
	`  if whichKey = t then put "That's the key"`
	`  -- more stuff...`
	The `keyDown` message is not sent for keys that generate special messages or for programmed function keys. A programmed function key would send a `functionKey` message and any additional messages that the specified function key is programmed to generate. See also the `functionKey` message description.

`doMenu` *var1*, *var2*	Sent to the current card when a menu item is selected. The parameter variable *var1* has the exact name of the menu item selected, including the three periods following menu items that invoke dialog boxes. Uppercase and lowercase don't matter, but you must type the three periods—don't use the Option-semicolon ellipsis character. The second parameter variable *var2* has the exact name of menu in the menu bar. (This message is also a HyperTalk command, which is listed in Chapter 10. An example handler to intercept the `doMenu` message is shown in the section "Redefining Commands," at the beginning of Chapter 10.)
`newBackground`	Sent to the current card as soon as a background has been created.

[continued]

■ **Table 8-3** Messages sent to the current card [continued]

Message	Meaning
`deleteBackground`	Sent to the current card when a background is deleted, just before it disappears.
`openBackground`	Sent to the current card when a background is first opened by going to a card whose background is different than that of the previous card.
`closeBackground`	Sent to the current card when a background is closed by going to another card that has a different background.
`newStack`	Sent to the current card when a stack is created.
`deleteStack`	Sent to the current card when a stack is deleted, just before it disappears.
`openStack`	Sent to the current card when a stack is opened by going to a card in a different stack than that of the previous card. In this case the following three messages are sent, in order: `openCard`, `openBackground`, and `openStack`.
`resumeStack`	Sent to the current card when you return to an already open stack.
`closeStack`	Sent to the current card when a stack is closed by opening another stack.
`suspendStack`	Sent to the current card when you switch to another stack.
`suspend`	Sent to the current card when HyperCard is suspended by launching another application with the `open` command just before the other application is launched. (Not sent under MultiFinder.)
`resume`	Sent to the current card when HyperCard resumes running after having been suspended. (Not sent under MultiFinder.)
`quit`	Sent to the current card when you choose Quit HyperCard from the File menu (or press Command-Q), just before HyperCard quits.

[continued]

■ **Table 8-3** Messages sent to the current card [continued]

Message	Meaning
`moveWindow`	Sent when you change a card window's `location` property with HyperTalk, drag or zoom the card window, or change the location of the card window with the `show` command. See also the `location` and `rectangle` properties in Chapter 12 and the `show` command in Chapter 10.
`hide menubar`	Sent to the current card when the menu bar is visible and you press Command–Space bar. (`Hide` is also a HyperTalk command; the command accepts other parameter variable values besides `menubar`; see its description in Chapter 10.)
`show menubar`	Sent to the current card when the menu bar is hidden and you press Command–Space bar. (`Show` is also a HyperTalk command; the command accepts other parameter variable values besides `menubar`; see the description in Chapter 10.)
`help`	Sent to the current card when Help is chosen from the Go menu (or Command-? is pressed). You can intercept this message if you want to provide your own Help system for your stack. (This message is also a HyperTalk command; see Chapter 10.)

Message order

For some events, HyperCard sends a sequence of system messages. The messages are sent in a specific message order. You can create message handlers that use the message sending order to set properties or perform other actions when opening stacks, creating new backgrounds, creating new cards, deleting cards, backgrounds, or stacks, moving from card to card, and moving from stack to stack. For example, the `startup` system message is always sent when HyperCard first starts up and the first stack is opened, so you can create a handler for the `startup` system message that sets properties for the first card before it appears.

You can create handlers for any message that is sent earlier in the message-sending order to change the actions that take place before the messages that follow it are acted upon by HyperCard. The HyperCard message-sending order that results from some of the actions most frequently performed when creating or modifying stacks is shown in Table 8-4. You can also open the message watcher to watch the messages that result from other actions.

■ **Table 8-4** HyperCard message-sending order

Action	Resulting message order
Start up HyperCard	`startUp`, `openStack`, `openBackground`, `openCard`
Resume HyperCard	`resume`, `openStack`, `openBackground`, `openCard`
Create a new stack	`closeCard`, `closeBackground`, `closeStack`, `newStack`, `newBackground`, `newCard`, `openStack`, `openBackground`, `openCard`
Create a new background	`closeCard`, `closeBackground`, `newBackground`, `newCard`, `openBackground`, `openCard`
Create a new card	`closeCard`, `newCard`, `openCard`
Delete a stack	`closeCard`, `closeBackground`, `closeStack`, `deleteStack`
Delete a background	`closeCard`, `closeBackground`, `deleteCard`, `deleteBackgroun`
Delete a card	`closeCard`, `closeBackground`, `deleteCard`, `deleteBackgroun` `openBackground`, `openCard`
Cut card	`closeCard`, `closeBackground`, `deleteBackground`, `openBackground`, `openCard`
Paste card	`newBackground`, `newCard`, `openBackground`, `openCard`

Chapter 9 Control Structures and Keywords

This chapter describes the nature of control structures and the control structure vocabulary, which is made up of HyperCard keywords.

A **keyword** is a word whose meaning is predefined in HyperTalk. You cannot redefine keywords as variable names. Keywords are not sent as messages when they execute in scripts, nor can they be used in the Message box (except for `do`, `if`, and `send`). Some keywords provide the structure for handlers; others control the flow of execution within handlers.

HyperTalk has two kinds of handlers: message and function handlers, denoted by the initial keywords `on` and `function`, respectively. Message and function handlers are defined in the same way (except for the initial keyword), but they differ in how they are invoked and in how they return values.

The syntax for each keyword is given. In the syntax statements, words in italic represent general elements that you replace with a specific instance when you write a statement; square brackets ([ ]) denote optional elements (don't type the square brackets).

Keywords in message handlers

The `on` keyword identifies a HyperCard message handler. Message handlers are written to define your own messages, or to modify or redefine what happens in response to any message (including a HyperTalk command). The general syntax of a message handler looks like this:

```
on messageName [parameterList]
    statementList
end messageName
```

MessageName is an identifier: a string starting with a letter and containing no spaces or punctuation marks except underscore; *parameterList* is a series of zero or more parameter variables (separated by commas if more than one); and *statementList* is one or more HyperTalk statements.

The handler dictates the method by which its object responds to *messageName*. When a message called *messageName* is sent to an object, HyperCard checks all of that object's message handlers to see if it has one named *messageName*. If so, the object responds according to that handler, and the message is sent no further (assuming the script has no `pass` statement, described later in this chapter). If the object has no handler to match *messageName*, HyperCard passes the message to the next object in the hierarchy.

- *You can override HyperTalk:* If you name a message handler the same as a built-in command, your name overrides the built-in one if yours is anywhere along the message-passing hierarchy between the object sending the message and HyperCard.

The statements in the handler execute until an `end`, `exit`, `pass`, or `return` keyword is reached (these keywords are discussed later in this section.) A message handler can return a value through the built-in function `the result` (discussed in Chapter 11).

On

`on` *messageName* [*parameterList*]

The `on` keyword marks the beginning of a message handler and connects the handler with a particular message. *MessageName* is the first word of the message to which the handler responds, and it is the name of the handler.

The optional *parameterList* allows the message handler to receive some values sent along with the message. This list is a series of local variable names, called *parameter variables*, separated by commas. When the message is sent, each source of value following the message name is evaluated; when the handler receives the message, each value is plugged into the parameter variable that appears in the corresponding position following `on` *messageName*. The first value in the list goes into the first parameter variable, and so on. If no values are sent with the message name, `empty` goes into each parameter variable.

Chapter 4, "Handling Messages," explains more about parameter passing. See also the `param`, `paramCount`, and `params` functions in Chapter 11, "Functions."

End

`end` *messageName*

The `end` keyword begins the last line of a handler—it is reached when all of the handler's statements have been executed (except for any bypassed conditional blocks). When the `end` statement is reached, the message that initiated execution of the handler is sent no further unless a `pass` *messageName* statement follows the `end` statement. (`Pass` is defined later in this section.) If the message that initiated this handler's execution was part of some other handler, control passes back to the other handler.

Exit

`exit` *messageName*

`exit to HyperCard`

The `exit` *messageName* statement ends execution of the handler.

The `exit to HyperCard` form makes program flow return directly to HyperCard, bypassing any pending handlers that have not finished executing.

Pass

`pass` *messageName*

The `pass` statement ends execution of the handler and sends the entire message that initiated execution of the handler to the next object in the hierarchy. (Ordinarily, a message is sent no further than the object containing the executing handler.)

Return

`return` *expression*

The `return` statement ends execution of the handler and, when it appears within a message handler structure, places the value of *expression* into the HyperTalk function `the result`.

The value of `the result` set by the `return` statement is valid only immediately after it executes; each new statement resets `the result`. (See the `result` function in Chapter 11, "Functions," for examples.)

Message handler example

The following example shows a handler that originates a message that in turn initiates execution of a second handler. (The second handler could be in the same script as the first or anywhere farther along the object hierarchy.)

```
on mouseUp
  heyNow 5 -- heyNow is the message name that's sent
end mouseUp
on heyNow timeVar -- Handler name heyNow matches message name
  play "boing"
  wait timeVar seconds
  play stop
  play "harpsichord" "ch d e f g a b c5w"
  play stop
end heyNow
```

Execution of the first handler is initiated when its object receives a `mouseUp` message. The `mouseUp` message could be generated when the user clicks the mouse or types `mouseUp` in the Message box and presses Return. It could also originate from another handler executing the statement `mouseUp` or could be sent explicitly to the handler's object with a `send` command.

When the `mouseUp` handler executes, it sends its one command statement (`heyNow 5`) as a message, first to its own object. The message name (the first word of the message) matches the handler name (the word following `on` in the first line of the handler), so the statements in the second handler begin executing. (If the current object had no `heyNow` message handler, that object would pass the entire message on to the next object in the hierarchy.)

The parameter value following `heyNow` in the first handler is passed into the parameter variable following `heyNow` in the second handler. So when the second handler is executing, `timeVar` has the value `5`.

Keywords in function handlers

A **function** is a named value that is calculated by HyperCard when a statement in which it's used executes. The `function` keyword identifies a HyperCard function handler. You can use this structure to define your own functions, which then can be called from any place in a statement where their values are needed. (User-defined functions are called like built-in HyperCard functions except that you must always use parentheses—see "Return," later in this section.)

Like message handlers, function handlers cannot be nested inside each other (or inside message handlers). The general syntax of a function handler looks like this:

```
function functionName [parameterList]
   statementList
end functionName
```

FunctionName is an identifier: a string starting with a letter and containing no spaces or punctuation marks except underscore; *parameterList* is a series of zero or more parameter variables separated by commas; and *statementList* is one or more HyperTalk statements.

User-defined function handlers use the message-passing hierarchy in the same way as do message handlers. That is, when the function name appears in a statement or in the Message box, HyperCard searches through all of the scripts along the current message-passing hierarchy for a matching function handler. If a match is found, the function handler executes. If none is found, the function call is passed to HyperCard; if there is no built-in function of that name, HyperCard displays an error dialog box.

If you name a function handler the same as a built-in function, your function is called when you use the function call syntax that uses parentheses. Of course, your function handler must also be in the script of an object lower or equal in the hierarchy than the originator of the function call. You can make calls to built-in functions using the function call syntax with `the` preceding the function name, which bypasses the message-passing hierarchy and calls HyperCard built-in functions directly.

Program flow runs through the function handler until it encounters an `end`, `exit`, `pass`, or `return` statement (discussed later in this section). A function handler returns a value directly into the statement in which its name was used.

Function

`function` *functionName*[*parameterList*]

The `function` keyword marks the beginning of a function handler and connects the handler with a particular function call. *FunctionName* is the function call to which the handler responds, and it is the name of the handler.

The optional *parameterList* allows the function handler to receive some values sent along with the function call. This list is a series of local variable names, called *parameter variables,* separated by commas. When the function call is made, each source of value appearing between parentheses following the function name is evaluated; when the handler begins to execute, each value is plugged into the parameter variable that appears in the corresponding position following `function` *functionName,* the first value in the list going into the first parameter variable, and so on.

For more details on passing parameters to function handlers, see "Return," later in this section.

End

`end` *functionName*

The `end` statement is the last line of the handler, reached when all of the handler's statements have been executed (except for any bypassed conditional blocks).

When the `end` statement is reached, control passes back to the handler containing the function call that originated the function handler's execution.

Exit

`exit` *functionName*
`exit to` HyperCard

The `exit` *functionName* statement ends execution of the handler.

The `exit to HyperCard` form makes program flow return directly to HyperCard, bypassing any pending handlers that have not finished executing, including the handler containing the function call.

Pass

`pass` *functionName*

The `pass` statement ends execution of the handler and sends the entire function call that initiated execution of the handler to the next object in the hierarchy. (Ordinarily, a function call is sent no further than the object containing the executing handler.)

Return

`return` *expression*

The `return` statement ends execution of the handler and, when it appears within a function handler structure, dictates the returned value of the function.

The value of *expression* replaces the function in the calling statement.

The function appears in the calling statement in the form *functionName*`(`*expressionList*`)`:

```
put square(17) into card field 1
```

ExpressionList is a series of zero or more expressions separated by commas whose values are assigned to the parameter variables in the *parameterList* of the function handler. In the above example, the *expressionList* comprises only the number `17`.

A user-defined function handler that would respond to the function call example `square(17)`, shown above, is

```
function square x
   return x * x
end square
```

In the example, the function handler has one parameter variable to receive one value passed to it by the calling statement. The value `17` is passed to the function handler where it is assigned to the parameter variable `x`; the value of `x * x` is returned by the `return` statement, replacing `square(17)` in the calling statement. So, the effect of the calling statement is to put the value `289` into card field 1.

- *Parentheses required:* User-defined functions are always followed by parentheses (which are empty if there are no parameters to pass). Unlike built-in functions (explained in detail in Chapter 11), user-defined functions can't be called with the form `the` *function* `of`.

Function handler example

The following function handler determines whether a number passed to it as a parameter is even or odd, returning the constant `true` if it's even or `false` if it's odd:

```
function evenNumber numberPassed
  return numberPassed mod 2 is 0
end evenNumber
```

A calling statement that would invoke the `evenNumber` function handler could be one like the following:

```
if evenNumber(numberVariable) then add 1 to evenNumberCount
```

In the calling statement, `numberVariable` can be the name of any variable or other source of value (including an actual number). HyperCard evaluates `numberVariable` before it passes the function call along the hierarchy, and its value is given to the parameter variable `numberPassed` when the `evenNumber` function handler is found. The part of the calling statement following `then` is arbitrary—the point of the example is to show how the function handler receives a value, examines it, and returns another value into the calling statement, based on the result of its execution.

Repeat structure

The `repeat` structure causes all of the HyperTalk statements between its first and last lines to execute in a loop until some condition is met or until an `exit` statement is encountered. The general syntax of a `repeat` structure looks like this:

```
repeat controlForm
   statementList
end repeat
```

ControlForm is one of the forms of the `repeat` statement described below, and *statementList* is any number of HyperTalk statements. `Repeat` structures can be used only within message handlers or function handlers.

◆ *Note:* If you want to try the examples in this chapter, be sure to put them within handlers.

Repeat statements

The `repeat` keyword begins the first line of a `repeat` structure. The repeat statement has five forms differentiated by the second word of the statement. Additionally, the `repeat with` form has two variations.

Repeat forever

```
repeat [forever]
```

The loop repeats forever, or until an `exit` statement is encountered:

```
put 1 into Message box
repeat
   add 1 to Message box
   if Message box contains 6 then exit repeat
end repeat
```

The example ends with `6` in the Message box.

For information on `exit repeat`, see "Exit Repeat Statement," later in this chapter. For information on `if`, see "If Structure," later in this chapter.

Repeat for

`repeat [for]` *number* `[times]`

Number is a source that yields a positive integer specifying how many times the loop is executed:

```
put 1 into Message box
repeat for 5 times
   add 1 to Message box
end repeat
```

The example ends with `6` in the Message box.

Repeat until

`repeat until` *condition*

Condition is an expression that always evaluates to `true` or `false`. The loop is repeated as long as the condition is false. The condition is checked prior to the first and any subsequent executions of the loop:

```
put 1 into Message box
repeat until Message Box contains 6
   add 1 to Message box
end repeat
```

The example ends with `6` in the Message box.

Repeat while

```
repeat while condition
```

Condition is an expression that evaluates to `true` or `false`. The loop is repeated as long as the condition is true. The condition is checked prior to the first and any subsequent executions of the loop:

```
put 1 into Message box
repeat while Message Box < 6
   add 1 to Message box
end repeat
```

The example ends with `6` in the Message box.

Repeat with

There are two variations of the `repeat with` form: one that increases a variable and one that decreases a variable.

`repeat with` *variable* = *start* `to` *finish*

Variable is a local or global variable name, and *start* and *finish* are sources of integers. The value of *start* is assigned to *variable* at the beginning of the loop, and is increased by 1 with each pass through the loop. Execution ends when the value of *variable* equals the value of *finish*.

```
repeat with increment = 1 to 6
   put increment into the Message box
end repeat
```

The example ends with `6` in the Message box. (This structure works much like a `FOR...NEXT` loop in BASIC.)

`repeat with` *variable* = *start* `down to` *finish*

The `down to` form is the same as the `to` form above, except that the value of *variable* is decreased by 1 with each pass through the loop. Execution ends when the value of *variable* equals the value of *finish*.

```
repeat with decrement = 6 down to 1
   put decrement into the Message box
end repeat
```

The example ends with `1` in the Message box.

Exit repeat statement

```
exit repeat
```

The `exit repeat` statement sends control to the end of the `repeat` structure, ending execution of the loop regardless of the state of the controlling conditions specified in the `repeat` statement.

```
put 1 into the Message box
repeat with increment = 1 to 100
  add increment to the Message box
  if Message box > 20 then
    beep 5
    exit repeat
  end if
end repeat
```

The example ends with `22` in the Message box.

An `exit` statement can appear anywhere within the structure.

For information on `if`, see "If Structure," later in this chapter.

Next repeat statement

```
next repeat
```

When a `next repeat` statement is encountered, control returns immediately to the top of the structure. (Usually, flow doesn't return to the top of the repeat structure until an `end` statement is encountered.)

```
repeat 20
  put random(9) into tempVar
  if tempVar mod 2 = 0 then next repeat
  put tempVar after field "oddNumbers"
end repeat
```

The example appends only the odd random numbers to the field, skipping any even ones.

A `next` statement can appear anywhere within the repeat structure.

For information on `if`, see "If Structure," later in this chapter. For more information about the `random` function, see Chapter 11.

End repeat statement

```
end repeat
```

The `end repeat` statement marks the end of the loop; it's the last line of a `repeat` control structure. When the controlling conditions specified in the `repeat` statement have been satisfied or an `exit` statement encountered, control goes beyond the `end` statement:

```
repeat for 5 times
   beep
end repeat
```

If structure

The `if` structure tests for the specified condition and, if the condition is true, executes the command statement or series of command statements that follow. The `if` structure has several forms, described in the following sections.

- *Note:* If you want to try the examples in this section, be sure to put them within handlers.

Single-statement If structure

A single-statement `if` structure has the form shown below:

`if` *condition* `then` *statement* [`else` *statement*]

A single-statement `if` structure can also occupy more than one line:

`if` *condition*
`then` *statement*
[`else` *statement*]

Within the `if` structure, *condition* is an expression that evaluates to `true` or `false`, and *statement* is a single HyperTalk command statement.

In the single-statement `if` structure, only one command statement can follow either `then` or `else` (if present), and the command statement must be on the same line with `then` or `else`.

If the condition between `if` and `then` is true, HyperCard executes the statement between `then` and `else` if `else` is present, or between `then` and the end of the line if `else` is not present following the statement, either on the same line or on the next line.

If the condition between `if` and `then` is false, HyperCard executes the statement between `else` and the end of the line if `else` is present, or it ignores the rest of the line if `else` is not present:

```
if Message box > 10 then beep 5 else beep 15
```

In this example, if the Message box holds a value greater than 10, the Macintosh beeps 5 times; if the value in the Message box is 10 or less, the Macintosh beeps 15 times.

◆ *Note:* Single-statement `if` structures can be used in the Message box.

Multiple-statement If structure

A multiple-statement `if` structure accommodates more than one executable statement following `then` and, optionally, more than one statement following `else`:

```
if condition then
   statementList
[else
   statementList]
end if
```

You can also end a multiple-statement `then` clause with a single-line `else`, in which case no `end if` statement is needed:

```
if condition then
   statementList
else statement
```

Condition is an expression that evaluates to `true` or `false`, and *statementList* is any number of HyperTalk statements.

In the multiple-statement `if` structure, more than one command statement can follow either `then` or `else` (if present), and the first command statement must be on the line following `then` or `else`. That is, if you want to have more than one statement in a block following `then` or `else`, put a return character after the respective `then` or `else`. Such a multiple-statement block must be ended explicitly: a multiple-statement `then` block can be ended with either `end if` or `else`; a multiple-statement `else` block must be ended with `end if`.

If the condition between `if` and `then` is true, HyperCard executes the statement or statements between `then` and `else` if `else` is present, or between `then` and `end if` if `else` is not present.

If the condition between `if` and `then` is false, HyperCard executes the statements between `else` and `end if` if `else` is present, or it ignores what's between `then` and `end if` if `else` is not present:

```
if number of this card is 10 then
   put "We're done!" into msg
   go Home
else
   put "And the next question is:" into msg
   go next card
end if
```

- *Note:* Multiple-statement `if` structures can be used only within message handlers or function handlers.

Nested If structures

`If` structures can be nested; that is, statements following a `then` or an `else` can include more `if` structures. Each nested multiple-line `if` structure must have its own `end if`, and an `else` always goes with the closest preceding `if` clause. The next example could be used as a `mouseUp` handler within a button.

```
repeat
  ask "Guess a number between 1 and 10:"
  if it is empty then
    exit repeat
  else
    if it is random(10) then
      put "You guessed it!"
    else
      put "Sorry, try again."
    end if
  end if
end repeat
```

In the example, satisfying the `if it is empty` condition in the first `if` structure allows the user to exit the loop by selecting the OK button without entering a number in the dialog box that is created with the `ask` command. Executing this example without the first `if` structure results in an endless loop, which you can exit by simultaneously clicking the Cancel button and pressing the Command-period keys.

- *Note:* Nested-statement `if` structures can be used only within message handlers or function handlers.

Do

`do` *expression*

The `do` keyword causes HyperCard to get the value of *expression;* HyperCard then sends that value as a message. (The `do` keyword in HyperTalk does not work like the `do` in Pascal does.)

In versions of HyperCard before HyperCard 2.0, if the *expression* was a field with more than one line, only the first line of the field was used by HyperCard and any lines that followed were ignored. HyperCard now evaluates any expression after the `do` keyword and sends it as a command. For example, if field 3 contains several lines of HyperTalk code and you say `do field 3`, the entire contents of field 3 are sent to HyperCard. Each line of the container executes just as if it were contained in a handler.

The `do` keyword can be used in the Message box.

Global

`global` *variableList*

VariableList is one or more HyperCard variable names separated by commas.

The `global` keyword makes a variable name known and its contents available to any script of any object in HyperCard. The following two lines are individual examples of `global` statements:

```
global myVar
global pages,sections,chapters
```

The following example handlers show a global variable being used for two handlers to access the same value:

```
on mouseUp
   global myVariable -- load the global here
   put 3 into myVariable
   writeResult
end mouseUp

on writeResult
   global myVariable -- can use the global as long as we define it here
   put myVariable -- the value remains 3
end writeResult
```

Changing the value of a global variable in any script changes its value everywhere. The `global` keyword must be used in each handler in which the global variable is used.

Global variables are not saved in between sessions of HyperCard or when HyperCard is suspended by launching another application with the `open` command.

Send

`send "`*messageName*`[`*parameterList*`]" [to`*object*`]`

MessageName is a string beginning with a letter and containing no spaces or punctuation marks other than underscore; *parameterList* is one or more expressions (separated by commas if more than one); and *object* is a HyperCard object descriptor or HyperCard. If no object is specified, HyperCard is the object.

The `send` keyword sends a message directly to a particular object, bypassing any handlers in the intervening message-passing hierarchy that might otherwise intercept the message.

```
send "hideIt" to field 3
send "addSums travel,food,hotel" to stack "expenseAccount"
send mouseUp to button "pushMe"
send "doMenu Print Card" to HyperCard
```

You can send a message directly to any object in the current stack or to another stack, but not to a specific object in another stack.

If the object has no message handler for *messageName*, the message is passed along the message-passing hierarchy stemming from the object to which the message was sent. If the object does have a matching handler, the handler executes, but the card to which it belongs does not necessarily open. Messages sent by executing the statements of the object's handler are sent along the receiving object's hierarchy.

Chapter 4, "Handling Messages," has more information about how the `send` command interacts with the message-passing hierarchy.

Quotation marks around the message are not required if the message is a single word, that is, only *messageName* and no parameters. Parameter expressions are evaluated before they are sent, even though the entire message has quotation marks around it.

`Send` does not change cards when a message is sent to an object on a card other than the current card, or cause HyperCard to send open or close messages to cards, backgrounds, or stacks. For example, if you are on the second card and send a message to a button on the ninth card in the stack, the ninth card doesn't get an `openCard` message.

- *You can use it in the Message box:* The `do`, `if`, and `send` keywords, unlike all other keywords, work in the Message box.

Chapter 10 **Commands**

This chapter describes all the commands in HyperTalk, showing their syntax and meaning.

HyperTalk commands are built-in message handlers that reside in HyperCard itself. When you issue a HyperTalk command, it's passed along the message-passing hierarchy as a message to HyperCard. In most cases there's no handler in any script along the way to intercept the message, so HyperCard receives the message and acts on it.

Some commands (such as `arrowKey`) are system messages as well as commands. This means two things: a system event can generate the message (pressing an arrow key generates the `arrowKey` message), and HyperCard has a built-in response to the message (`arrowKey` takes you to another card).

Redefining commands

You can write a message handler that redefines a built-in command. Redefining commands is especially useful for trapping menu commands you want to modify or that you want to prevent a user from issuing (for example, `on doMenu` *menuItem*).

Be wary, however: once a command—or any message—has been intercepted by a handler, it's sent no further along the hierarchy; so your newly defined command replaces the HyperCard built-in one. If, for example, you write a handler for the `doMenu` command, be sure to pass the message if you don't want to prevent *every* instance of it from reaching HyperCard:

```
on doMenu menuItem
  if menuItem is "Delete Card" then
    answer "Are you sure?" with "Delete" or "Cancel"
    if It is not "Delete" then exit doMenu
  end if
  pass doMenu
end doMenu
```

If you inadvertently fail to pass `doMenu`, you may find yourself apparently unable to use any menu command, even to fix the `doMenu` handler. (In that case, from the Message box, execute the command `edit script` for the object containing the handler. If the Message box is hidden and blind typing is false, go to the last card of the Home stack and turn blind typing on.)

Syntax description notation

The syntax descriptions use the following typographic conventions. Words or phrases in `this font` are HyperTalk language elements or are those that you type to the computer literally, exactly as shown. Words in *italics* describe general elements, not specific names—you must substitute the actual instances. Square brackets ([ ]) enclose optional elements that may be included if you need them. (Don't type the square brackets.) In some cases, optional elements change what the message does; in other cases they are helper words that have no effect except to make the message more readable.

It doesn't matter whether you use uppercase or lowercase letters; names that are formed from two words are shown in lowercase letters with a capital in the middle (`likeThis`) merely to make them more readable. The HyperTalk prepositions `of` and `in` are interchangeable—the syntax descriptions use the one that sounds more natural.

The term *yields* indicates a specific kind of value, such as a number or a text string, that must result from evaluation of an expression when a restriction applies. (For example, the expression and the destination in an `add` command must yield numbers.) However, any HyperTalk value can be treated as a text string.

Some of the syntax statements and examples in this chapter use the soft return (¬) character to continue long statements onto the next line. The soft return is used here because of the line length limitations of the page format used in this chapter. You should avoid using soft returns in your scripts so that the statements in your handlers are easier to read.

To help you quickly locate commands that are new or enhanced since the first edition of this book, the names of the new and enhanced commands are followed with a symbol like the one at the beginning of this paragraph.

Add

Syntax

`add` *number* `to [`*chunk* `of]` *container*

Number is an expression that yields a number. *Chunk* is an expression that yields a chunk of a container. *Container* is an expression that yields a field, the Message box, the selection, or a variable.

Examples

```
add 3 to It
add field "Amount" to field "Total"
```

Description

The `add` command adds the value of *number* to the value of *container*, and leaves the result in *container*. The value previously in the container must be a number; it is replaced with the new value.

Script

The following example handler sums numbers in a field, if each line of the field contains one number, and puts the result into the Message box. The name of the field is passed to the handler as a parameter.

```
on sumField whichField
  put 0 into total
  repeat with count = 1 to the number of lines in whichField
    add line count of whichField to total
  end repeat
  put total into msg
end sumField
```

Answer

Syntax

```
answer question [with reply [or reply2 [or reply3]]]
answer file [promptText] [of type fileType]
```

Question and *reply* are expressions that yield text strings. *Question* can be up to 6 lines (255 characters total). *PromptText* is an expression that yields a prompt text string. *FileType* is one of the following literal file types: `stack`, `text`, `application`, `picture`, `paint`, or `painting`. You can also specify the same file types with the following Macintosh four-letter file-type designators: `STAK`, `TEXT`, `APPL`, `PICT`, and `PNTG`.

Examples

```
answer "Which is the way the world ends?" with "Bang" or¬
"Whimper"
answer myQuestion with myAnswer or field 7
answer file "Pick a text file:" of type TEXT
answer file "" of type PICT
```

Description

The `answer` command displays a dialog box with a question and up to three buttons, each representing a different reply.

The dialog box stays on the screen until one of the buttons is clicked; pressing Return or Enter has the same effect as clicking the button farthest to the right, which correlates to the last reply specified with the `answer` command.

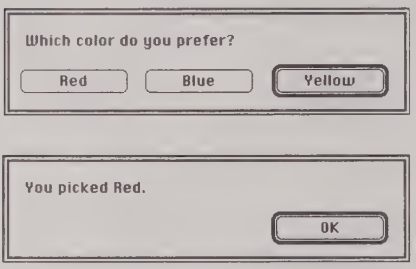
The `answer file` form of the `answer` command displays a directory dialog box for opening files. You can use the optional `of type` *fileType* form of the command to notify HyperCard of the type of files to display in the dialog box. You can also supply a default user prompt by supplying a quoted string for the *promptText* form of the command. The name of the selected file is placed in the container `It`.

Script

The following example handler produces the dialog boxes in Figure 10-1. (The second one depends on which button you click in the first one.)

```
on chooseColor
   answer "Which color do you prefer?" with "Red" or "Blue"¬
   or "Yellow"
   if It is "Red" then answer "You picked Red."
   else if It is "Blue" then answer "You picked Blue."
   else if It is "Yellow" then answer "You picked Yellow."
end chooseColor
```

■ **Figure 10-1** `Answer` command dialog boxes



The following example handler queries the user for the name of a document to print and an application with which to print it:

```
on mouseUp
   answer file "Print what document?" of type TEXT
   if It is not empty then
      put It into doc
      answer file "Use what application?" of type application
      if It is not empty then print doc with It
   end if
end mouseUp
```

Notes

There is no way for a script to reply to a dialog box by itself, so it's important that a script meant to run unattended not use `answer`.

The text of the button clicked goes into the local variable `It`. If no reply is specified, HyperCard displays one button containing OK.

Neither the question nor any of the replies can have more than one line. If you use a container that has more than one line of text in it for the question, only the first line appears. If you use a container with more than one line for a reply, the last line is displayed in the button. (Only the center portion shows if the line is too long to fit in the button.) However, all lines go into the local variable `It` when the button is clicked.

Unless you're using container names, put the question and the replies inside quotation marks if they contain any spaces.

Each reply can be up to 13 characters long (depending on the width of the characters).

If you do not supply a file type for the `answer file` form, all file types are displayed in the dialog box. If you do not supply text for a prompt, no default prompt is displayed by the dialog box.

See also the `ask` command, later in this chapter.

ArrowKey

Syntax

`arrowKey` *direction*

Direction is an expression that yields the name of one of the arrow keys: `left`, `right`, `up`, or `down`.

Examples

```
arrowKey left
arrowKey down
```

Description

The `arrowKey` command takes you to another card. The effects of the `arrowKey` command are shown in Table 10-1.

■ **Table 10-1** Effects of the `arrowKey` command

Parameter value	Effect
`left`	Go to previous card in current stack
`right`	Go to next card in current stack
`up`	Go forward through recent cards
`down`	Go back through recent cards

The `arrowKey` message, which invokes the `arrowKey` command if it reaches HyperCard, is normally generated by pressing any of the arrow keys on the keyboard. (Which arrow key you press determines the message's parameter value.) You can also send `arrowKey` from the Message box or execute it as a line in a script.

Script

The following example handler works with extended keyboards. It makes function keys 9, 10, 11, and 12 send the `arrowKey` message with parameters of `left`, `right`, `up`, and `down`, respectively:

```
on functionKey whichKey -- map function keys to arrow keys
  if whichKey is 9 then arrowKey left
  else if whichKey is 10 then arrowKey right
  else if whichKey is 11 then arrowKey up
  else if whichKey is 12 then arrowKey down
end functionKey
```

Notes

The `textArrows` property alters the effect of pressing the arrow keys (see "TextArrows" in Chapter 12), but it does not affect the `arrowKey` command.

See also the `arrowKey` message in Table 8-3.

Ask

Syntax

```
ask question [with defaultAnswer]
ask password [clear] question [with defaultAnswer]
ask file [promptText] [with [default] fileName]
```

Question and *defaultAnswer* are expressions that yield text strings. *PromptText* is an expression that yields a prompt text string. *FileName* is an expression that yields a default filename to be displayed in the filename field of the dialog box.

Examples

```
ask "Who needs this kind of grief?" with "Not me."
ask field 1 with line 1 of field 2
ask password "Please enter your password:"
ask file "Save this file as:" with "MyTextFile"
ask file ""
```

Description

The `ask` command displays a dialog box containing a question with a text box into which the user can type an answer. The optional *defaultAnswer* string specifies an answer that appears initially in the window, highlighted so it can be easily replaced. The dialog box appears with OK and Cancel buttons.

The `ask password` form causes the answer to be encrypted as a number (which is placed into the local variable `It`). The encrypted answer can be stored in a field to be compared to a later answer to `ask password` if, for example, you want the user to be able to protect data entered into the stack. Password protection built into a stack in this manner is separate from that set up by the Protect Stack command in the File menu. The text entered in the password dialog box is hidden from the user.

The `ask file` form of the `ask` command displays a standard dialog box for saving files. You can use the optional `default` *fileName* form of the command to place a filename in the filename text entry field of the dialog box. The *fileName* string should be 23 characters or less in length to fit into the text entry field. The default filename is replaced by the user when the user starts typing. If you do not supply a default filename, the filename text entry field is empty when the dialog box is displayed.

6

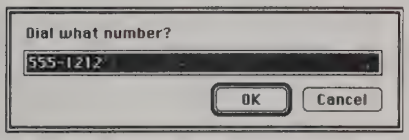
You can also supply a default user prompt by using the *promptText* form of the command. The *promptText* string should be 26 characters or less in length to fit into the dialog box. If you do not supply text for a prompt, no default prompt is displayed in the dialog box.

Script

The following example handler produces the dialog box in Figure 10-2:

```
on phone
   ask "Dial what number:" with "555-1212"
   if It is not empty then dial It
end phone
```

■ **Figure 10-2** `Ask` command dialog box



Notes

The text typed into the box (the answer to the question, the password, or the filename) goes into the local variable `It`, either when the OK button is clicked or when Return or Enter is pressed. If the Cancel button is clicked, the dialog box goes away, but the answer is not placed into `It`.

If you use the `ask password clear` form of the `ask password` command, the password is not encrypted when it is typed in the dialog box.

The question and default answer can both be up to six lines in a container with a maximum of 255 characters. Unless you're using container names, put the question and the default answer inside quotation marks if they contain any spaces (or if, as in the example, they are telephone numbers containing a hyphen—to prevent HyperCard from doing subtraction).

See also the `answer` command, earlier in this chapter.

Beep

Syntax

`beep [`*number*`]`

Number is an expression that yields an integer.

Examples

```
beep 5
beep line 3 of field 1
```

Description

The `beep` command causes the Macintosh speaker to make a beep sound *number* times. If no *number* is given, the speaker beeps once.

Script

The following example handler uses the `beep` command to alert the user that an `answer` dialog box, to which the user must reply, is being displayed:

```
on openStack
   beep 2
   answer "Do you need instructions?" with "Yes" or "No"
   if It is "Yes" then go to stack "Instructions"
end openStack
```

Choose

Syntax

```
choose toolName tool
choose tool toolNumber
```

ToolName is an expression that yields the name of any one of the tools from the HyperCard Tools palette (shown in Figure 10-3). *ToolNumber* is an expression that yields an integer from 1 to 18.

Examples

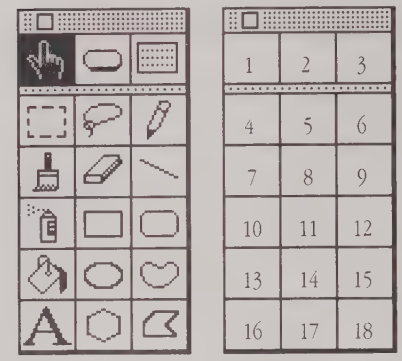
```
choose browse tool
choose tool 11
choose eraser tool
```

Description

The `choose` command changes the current tool as though you had chosen it from the Tools palette. Valid tool names and numbers are

```
browse (1)          oval (14)
brush (7)           pencil (6)
bucket (13)         poly[gon] (18)
button (2)          rect[angle] (11)
curve (15)          reg[ular] poly[gon] (17)
eraser (8)          round rect[angle] (12)
field (3)           select (4)
lasso (5)           spray [can] (10)
line (9)            text (16)
```

■ **Figure 10-3** Tools palette



Script

The following example shows a typical use of the `choose` command in a handler:

```
on drawBox
   reset paint
   choose rectangle tool
   set lineSize to 2
   drag from 50,50 to 150,150
   choose browse tool
end drawBox
```

Notes

You must have the user level set to Painting, Authoring, or Scripting to use the `choose` command, but the Tools palette need not be visible.

You can get the name of the current tool by using the function `the tool` in a handler or the Message box.

Setting user levels is described in the *HyperCard Reference* and in the `userLevel` global property description in Chapter 12. The `tool` function is described in Chapter 11.

Click

Syntax

```
click at point [with key [,key2 [,key3 ]]]
```

Point is an expression yielding a point: two integers separated by a comma, representing horizontal and vertical pixel offsets (respectively) from the upper-left corner of the card window. *Key, key2,* and *key3* are expressions that yield one of the following key names, separated by commas: `shiftKey`, `optionKey`, or `commandKey` (or `cmdKey`).

Examples

```
click at 100,100
click at the loc of button "Press me" with optionKey
```

Description

The `click` command causes the same actions that would result from clicking with the pointer at the specified point on the screen: the system messages `mouseDown` and `mouseUp` are sent to the objects under the pointer (but the visible pointer is not moved from its current location).

Using a `with` *key* form produces the same result as clicking the mouse button while holding down the specified key (or keys).

If *point* is within an unlocked field, the insertion point is set: if there is text at or past *point,* the insertion point is set at *point;* if there is text on the same line as *point* but *point* is beyond the end of text, the insertion point is set at the end of text on that line; if there is no text at *point,* the insertion point is set at the start of the line.

You can select a word by double-clicking it (that is, by executing the `click` command twice in succession at the location of the word). You can select any string of text by using `click` at the beginning then using `click with shiftKey` at the end of the string.

Script

The following example handler selects and displays a word from a locked field when you click on the word (`mouseUp` is not sent to unlocked fields when you click them):

```
on mouseUp
   set lockText of me to false
   click at the clickLoc
   click at the clickLoc
   get the selection
   put It into the Message box
   set lockText of me to true
end mouseUp
```

Notes

The pixel offset values of *point* are not restricted to the size of the screen, but are misinterpreted if greater than `32767`.

See also the `drag` command, later in this chapter.

Close file

Syntax

`close file` *fileName*

FileName is an expression that yields a text string that is a valid filename.

Examples

```
close file myData
close file "myDisk:myFolder:myFile"
```

Description

The `close` command closes a disk file previously opened with the `open file` command to import or export ASCII text. The expression *fileName* must yield a valid Macintosh filename, including pathname if required.

Script

The following example handler reads any size text file into a global variable named `temp`:

```
on importText
  global temp
  put "MyFilename" into filename
  open file filename
  repeat
    read from file filename for 16384
    if It is empty then exit repeat
    put It after temp
  end repeat
  close file filename
end importText
```

Notes

If the specified file is not open, an error is generated. The error is stored in the HyperCard function `the result`. Use the `close file` command to close files explicitly after you use them. HyperCard automatically closes all open files when an `exit to HyperCard` statement is executed, when you press Command-period, or when you quit HyperCard.

You must provide the full pathname of the file if it's not at the same directory level as HyperCard. (See "Identifying a Stack" in Chapter 5 for an explanation of pathnames.)

See also the `open file`, `read`, and `write` commands, later in this chapter and the `result` function in Chapter 11.

Close printing

Syntax

```
close printing
```

Description

The `close printing` command ends a print job previously begun with the `open printing` command or the `open report printing` command. Nothing is actually printed until the `close printing` command is executed.

Script

The following example handler executes a printing job. It prints a specified number of cards, beginning on a specified card:

```
on printRange low,high
  push card
  open printing
  go to card low
  print (high-low) + 1 cards
  close printing
  pop card
end printRange
```

Note

See also the `open printing`, `open report printing`, `print`, and `print card` commands, later in this chapter.

Close window

Syntax

`close window` *windowName*

WindowName is an expression that yields the name of a window.

Description

The `close window` command closes the specified stack. `Close window` works on stack windows only if more than one stack is open. The `close window` command also works on stacks that are hidden. When the specified window is closed, a `close` system message is also sent by HyperCard.

Example

```
close window "Planets"
```

Note

See also the `go` command in this chapter and the `close` system message described in Chapter 8, "System Messages."

ControlKey

Syntax

`controlKey` *KeyNumber*

KeyNumber is an expression that yields an integer between 1 and 127. The values 97 through 126 do not correspond to any key.

Example

```
controlKey 26
```

Description

The `controlKey` command passes a numeric value, *keyNumber*, which represents a key pressed in combination with the Control key.

Script

The following example handler in a stack script in the current message-passing hierarchy prints the current card when it intercepts the `controlKey` system message and the parameter `16`. The handler works if either the Control-P key combination is pressed or you type the command `controlKey 16` in the Message box.

```
on controlKey whichKey
  if whichKey = 16 then
    doMenu "Print Card"
    exit controlKey
  end if
  pass controlKey
end controlKey
```

Notes

`ControlKey` is also a system message sent to the current card when the Control key is pressed in combination with another key. The acceptable values for the *keyNumber* parameter are shown in Table 10-1.

There are no built-in meanings for Control-key combinations in HyperCard. Nothing happens when a given Control-key combination is pressed or a `controlKey` command is executed unless you have a handler for that Control-key combination in the script of the current card or in a script somewhere in the message-passing hierarchy between the current card and HyperCard.

See also the `controlKey` system message in Table 8-3.

■ **Table 10-1** `ControlKey` message parameter values

Parameter value	Keys	Parameter value	Keys
1	a, Home	27	Esc, Clear, Left-bracket ([ )
2	b	28	Backslash (\), Left Arrow
3	c, Enter	29	Right-bracket ( ]), Right Arrow
4	d, End	30	Up Arrow
5	e, Help	31	Hyphen (-), Down Arrow
6	f	39	Single Quotation Mark (')
7	g	42	Asterisk (*)
8	h, Delete	43	Plus (+)
9	i, Tab	44	Comma (,)
10	j	45	Minus (–)
11	k, Page Up	46	Period (.)
12	l, Page Down	47	Slash (/)
13	m, Return	48	0
14	n	49	1
15	o	50	2
16	p, all function keys	51	3
17	q	52	4
18	r	53	5
19	s	54	6
20	t	55	7
21	u	56	8
22	v	57	9
23	w	59	Semicolon (;)
24	x	61	Equal (=)
25	y	96	Tilde (~)
26	z	127	Forward Delete

Convert

Syntax

`convert [chunk of]` *container* `to` *format* `[and` *format*`]`

Container is an expression that yields a container name and *format* is an expression that yields a time or date format specification.

Examples

```
convert timeVariable to seconds
convert line 1 of second card field to long date and short time
```

Description

The `convert` command converts a date or time in the specified container to the format specified. The optional second format specification is used when a date and time are both included. Valid format specifications and their meanings are as follows:

`seconds`	Seconds since midnight, January 1, 1904.
`dateItems`	A comma-separated list of numbers representing (in order): year, month, day, hour, minute, second, and day of week.
`long date`	The date in text form: `Tuesday, October 17, 1989`.
`short date`	The date in slash-separated numeric form: `10/17/89`. The date separators may be different in countries other than the United States.
`abbreviated date`	The date in text form with abbreviated day of week: `Tue, October 17, 1989`.
`long time`	The time in colon-separated form including seconds: `5:15:15 PM`.
`short time`	The time in colon-separated form without seconds: `5:15 PM`.
`abbreviated time`	Same form as short: `5:15 PM`.

Script

The following example handler counts the seconds elapsed while a command in the Message box executes:

```
on mouseUp
   put the long time into startTime
   convert startTime to seconds
   if msg is not empty then do msg
   put the long time into endTime
   convert endTime to seconds
   answer "That took" && endTime - startTime && "seconds."
end mouseUp
```

Notes

If you try to convert an invalid date, for example, Friday, May 23, 1990, HyperCard displays a dialog box with the error message "Invalid date." See also, the `result` function in Chapter 11, "Functions."

The modifier `abbreviated` can be shortened to `abbrev` or `abbr`. If you change the date format from the Control Panel, the order of the items of the date format may be different from what is shown in the short date description. The `dateItems` result is always in the same order regardless of the setting in the Control Panel.

Create menu

Syntax

```
create menu menuName
```

MenuName is an expression that yields the name of a new menu to be added to the menu bar.

Examples

```
create menu "Home"
create menu "Vacation locations"
```

Description

You use the `create` command to add a new menu to the HyperCard menu bar. After the new menu is created, it remains in the menu bar until it is deleted with the `delete` command, or you use the `reset number` command, or you quit HyperCard. If you want a set of menus to remain in the menu bar while a certain stack is open, create your menus in that stack's `openStack` system message handler. You can delete or disable your menus when you close your stack or open another stack by using the `delete` or `disable` commands in `closeStack` or `suspendStack` system message handlers. See Chapter 8 for more information about system messages.

You can put menu items into the new menu with the `put` command, described later in this chapter. You can also specify a menu message to be sent when a menu item is chosen.

Notes

See also the `checkMark`, `cmdChar`, `enabled`, `markChar`, `menuMsg`, `name`, and `textStyle` properties in Chapter 12 "Properties."

See also the `delete`, `disable`, `enable`, and `put` commands in this chapter.

Create stack

Syntax

`create stack` *stackName* `[with` *background*`] [in a new window]`

StackName is an expression that yields the name you want to assign to the new stack. *Background* is an expression that yields the descriptor of a background you want to use in the new stack. The background must be one in the current stack.

Examples

```
create stack "Ghosts" with this background
create stack "Mystery" with bkgnd 3 of this stack
```

Description

The `create` command creates a new stack with the specified name and optionally with a specified background of the current stack.

If you do not specify a background, the stack is created with a blank background. If you use the form `create stack` *stackName* `in a new window`, the stack is created, appears in another window, and becomes the current stack. If you do not use the form `in a new window`, the current stack is closed.

Notes

To create a stack with a background in a stack other than the current stack, you must go to the stack with the background you want to use before using the `create stack` command.

If the `create stack` command generates an error, it is stored in the HyperCard function `the result`.

The new stack is created with the same card size as the current stack. You can change the card size by resetting the `rectangle` property of the `card window` or by clicking the Resize button in the Stack Info dialog box, and then dragging the lower-right corner of the representation of the card to the size you want.

See also the `result` function in Chapter 11 and the `rectangle` property in Chapter 12.

Debug checkpoint

Syntax `debug checkpoint`

Example `debug checkpoint`

Description The `debug checkpoint` command sets a checkpoint in a HyperTalk handler. When HyperCard encounters this message in a handler it enters the debugger: it pauses execution of the handler and opens the script editor window, putting a box around the line where the checkpoint is set.

Once you are in the debugger, you can step or trace through the remaining lines of the script. You can step by pressing Command-S or choosing Step from the Debugger menu. You can step into by pressing Command-I or choosing Step Into from the Debugger menu. You can trace by choosing Trace from the Debugger menu. You can trace into by pressing Command-T or choosing Trace Into from the Debugger menu. With each step, the next line of HyperTalk in the current handler is selected and executed.

Notes In combination, checkpoints and stepping can be used to examine, diagnose, and modify the behavior of sophisticated HyperTalk handlers. Checkpoints only work at user level five.

See also Chapter 3, "The Scripting Environment," for more information about the debugger environment.

Delete

Syntax

```
delete chunk [of container]
delete menu
delete menuItem of menu
```

Chunk is a chunk expression referring to some text in a specified field and *container* specifies a container. *Menu* is an expression that yields a menu descriptor. *MenuItem* is an expression the yields a menu item descriptor.

Examples

```
delete line 1 of field 1
delete char 1 to 5 of line 4 of field "Charlie" of second card
delete menuItem "Paths..." of menu "Home"
delete menu "Windows"
delete second menuItem of menu 6
```

Description

The `delete` command removes specified text from the designated container in the current stack, a specified menu item, or a specified menu.

You can delete the Tools, Patterns, Font, or Apple menus, but you can't delete the menu items within those menus. That is, even though the menus are not visible in the menu bar, the menu items in those menus are not disabled. They can still be activated with the `doMenu` command or with their Command-key equivalents.

Script

The following example handler finds and deletes a name from a list with one name per line:

```
on zapaName
   put "Smith" & return & "Jones" & return & "Taylor"¬
   into list
   ask "Delete which name from the list?" with empty
   repeat with count = the number of lines in list down to 1
      if It is in line count of list then delete line count of¬
      list
   end repeat
end zapaName
```

Notes

Using the `delete` command to delete a chunk is not the same as using `put empty into` with the same chunk of text specified. For example, if you delete a line in a field with a statement like

```
delete line 4 of field 7
```

you delete the return character as well as the text; what was previously the fifth line becomes the fourth. The following statement leaves the return character in line 4:

```
put empty into line 4 of field 7
```

Even if you delete all of the text in a field, the field remains defined on the card or background, unlike selecting the field and choosing Cut Field or Clear Field from the Edit menu.

When you delete text in a field on a card other than the current one, the current card does not change. If you delete the text in a background field that has the `sharedText` property set to `true`, the text in that background field is deleted on every card with that background field.

In the case of the HyperCard standard menu items, the `doMenu` command works even when the item is deleted with the `delete` command. For example, if the File menu is deleted and the following handler is executed, HyperCard exits to the Finder:

```
on mouseDown
   delete menuItem "Quit HyperCard" from menu "File"
   doMenu "Quit HyperCard"
end mouseDown
```

Command-key equivalents, however, do not work for menu items that have been deleted, with one exception: Command-Q still works after the Quit HyperCard menu item is deleted.

Chapter 5 describes how to designate menus and menu items. Chunk expressions are described in Chapter 6. See also the `create menu`, `disable`, `enable`, and `put` commands in this chapter.

Dial

Syntax

`dial` *number* `[with modem [`*modemCommands*`]]`

Number is an expression that yields a string with numbers in it and *modemCommands* are commands for your modem.

Examples

```
dial steve -- if steve is a variable containing a number
dial "415-555-1212"
dial "493-996-1010" with modem "ATS0=0S7=1DT"
dial "493-973-6000" with modem
```

Description

The `dial` command, without the `with modem` option, generates the touch-tone sounds for the digits in *number* through the Macintosh speaker. Holding the telephone handset up to the speaker works on some telephones; for others you need a device that feeds the Macintosh audio output to the telephone.

If you use the `with modem` option, the `dial` command sets up telephone calls using the Apple Modem 300/1200, the Apple Personal Modem, or any Hayes-compatible modem attached to the Macintosh serial port. The *modemCommands* parameters are those described in the manual for your modem. Their default value is `"ATS0=0DT"`.

If *number* yields a string with numbers including a hyphen (as in 555-1212), enclose it within quotation marks to prevent HyperCard from doing subtraction with the hyphen before passing the number to the `dial` command (which ignores characters other than numbers). Similarly, enclose the *modemCommands* parameter within quotation marks.

Notes

You can press Command-period to exit the `dial` command during the ten-second wait imposed when dialing with a modem.

Disable

Syntax

```
disable menu
disable menuItem of menu
```

Menu is an expression that yields a menu descriptor. *MenuItem* is an expression the yields a menu item descriptor.

Examples

```
disable menu "Home"
disable menu 5
disable menuItem "Get Back" of menu "Direction"
disable the fourth menuItem of sixth menu
```

Description

The `disable` command disables a specified menu or menu item. When you create a menu item it's default state is enabled. When a menu or menu item is disabled, it is gray and inactive. The `disable` command also sets the `enabled` property to `false`.

If you `disable` a menu item that does not exist, HyperCard displays a "No such menu item" dialog box. If you `disable` a menu that does not exist, HyperCard displays a "No such menu" dialog box.

Notes

See also the `cmdChar`, `enabled`, `markChar`, and `menuMsg` properties in Chapter 12, "Properties," and the `create menu`, `enable`, and `put` commands in this chapter.

Except for Command-Q, Command-key equivalents do not work on menu items that have been deleted or disabled.

Divide

Syntax

`divide [`*chunk* `of]` *container* `by` *number*

Chunk is an expression that yields a chunk expression. *Container* is a container that holds a numeric value, and *number* is any expression that yields an numeric value.

Examples

```
divide field "total" by 3
divide fahrenheit by celsius  -- if fahrenheit and celsius are
                              -- variables
divide line 3 of field 4 by 10
```

Description

The `divide` command divides the value of `[chunk of]` *container* by the value of *number* and puts the result into *container*.

Script

The following example handler figures the percentage represented by a fraction of two numbers specified as parameters:

```
on percent var1,var2
  divide var1 by var2
  put trunc(var1 * 100) & "%"
end percent
```

Notes

The value previously in the container must be a number; it is replaced with the new value.

Division by 0 puts the result `INF` into *container*. (`INF` is the SANE value for infinity.) Division is carried out to a precision of up to 19 decimal places. The value for the amount of precision is set with the `numberFormat` property.

See also the `numberFormat` global property in Chapter 12 and the discussion of numbers in Chapter 6.

Do

Syntax

`do` *expression*

Expression is any valid HyperTalk expression, a handler name, container, or chunk of a container.

Examples

```
do "MyHandler"
do background field 2
do first word of line 3 of card field 1
do line five of card field 2
```

Description

The `do` command causes HyperCard to get the value of *expression*, then sends it as a message. (The `do` command in HyperTalk does not work like the `do` in Pascal does.) `Do` is also a HyperCard keyword.

If a field contains several lines of HyperTalk statements, you can send the contents of that field with the `do` command. For example, if field 3 contains several lines of HyperTalk code and you say `do field 3`, the entire contents of field 3 is sent to HyperCard. Each line of the field executes just as if it were contained within a handler.

The `do` command can be used in the Message box.

DoMenu

Syntax

```
doMenu itemName [menuName] [without dialog]
```

ItemName is an expression that yields the name of a menu command.

Examples

```
doMenu "open stack..."
doMenu "Control Panel" -- desk accessory in the Apple menu
```

Description

The `doMenu` command performs the menu command specified by *menuItem* as though you had chosen the command directly from the appropriate HyperCard menu. The `without dialog` form of the `doMenu` command performs the specified menu command action without providing a dialog box.

Script

The following example handler checks for a `doMenu` message with the Quit HyperCard menu item and puts up a dialog box when the Quit HyperCard menu item is selected:

```
on doMenu menuChoice
  if menuChoice is "Quit HyperCard"
  then
    answer "Are you sure you want to Quit" ¬
    with "OK" or "Cancel"
    if it is "Cancel"
    then
     exit doMenu
    else
     pass doMenu
    end if
  else
    pass doMenu
  end if
end doMenu
```

Notes

Both the specified menu item and the menu in which it resides must be available at the current user level (as described in the *HyperCard Reference*). If there are periods following the menu item, you must include them in *menuItem*. (You can't use the ellipsis character in their place.) For example, `"Open Stack..."` is a HyperCard menu item with three periods that follow. You don't have to specify which menu the menu item comes from.

Be aware that some menu commands change with conditions. (For example, Paste Card can change to Paste Button, depending on the contents of the Clipboard.)

- *Don't lock yourself out:* If you write a handler to intercept `doMenu`, be sure to pass the message after examining the new menu item. (Notice that the example passes `doMenu` twice, to satisfy the conditions for each `if` statement.) Otherwise, you may find yourself apparently unable to use any menu command, even to fix the `doMenu` handler or to quit HyperCard. (In that case, from the Message box, execute the command `edit script` for the object containing the handler. If the Message box is hidden and blind typing is false, go to the last card of the Home stack and turn blind typing on.)

Drag

Syntax

`drag from` *point1* `to` *point2* `[with` *key* `[,`*key2*`[,`*key3*`]]]`

Point1 and *point2* are expressions, each of which yields a point: two integers separated by a comma, representing horizontal and vertical pixel offsets (respectively) from the upper-left of the Macintosh screen. *Key*, *key2*, and *key3* are one of the following key names, separated by commas: `shiftKey`, `optionKey`, or `commandKey` (or `cmdKey`).

Examples

```
drag from 100,100 to 200,200
drag from the loc of button 1 to the mouseLoc with ¬
commandKey,shiftKey
```

Description

The `drag` command performs the same action as though you had dragged manually, except that in order to select text in a field using the `drag` command, you must use `with shiftKey`. In all other cases, using the `with` *key* form produces the same result as dragging while holding down the specified key.

Script

The following example handler draws random-sized ovals filled with random patterns on a new card:

```
on mouseUp
   doMenu "New Card" -- so we don't draw on current card
   choose oval tool
   set filled to true
   repeat until the mouseclick
      set pattern to random(30)
      drag from random(319),random(199) to random(319),¬
      random(199)
   end repeat
   choose browse tool
   doMenu "Delete Card" -- get rid of the card we just made
   go previous card -- take us back where we started from
end mouseUp
```

Notes

You can use `drag` with any tool selected, but it has no effect with some Paint tools.

The location of the actual pointer doesn't change from where it was before the command was issued.

See also the `click` command earlier in this chapter, and the `dragSpeed` property in Chapter 12.

Edit script

Syntax

`edit script of` *object*

Object is a expression that yields a descriptor of an object: a stack, card, background, field, or button.

Examples

```
edit script of button 1
edit script of this stack
```

Description

The `edit script` command opens the script of the specified object with the HyperCard script editor and pauses script execution until the script window is closed or the card window is brought to the front.

Script

The following example handler enables you to edit the script of any button or field merely by positioning the pointer over it and pressing the Option key:

```
on mouseWithin
  if the optionKey is down then edit script of the target
end mouseWithin
```

Note

Refer to Chapter 3, "The Scripting Environment," for an explanation of how the script editor works.

Enable

Syntax

```
enable menu
enable menuItem of menu
```

Menu is an expression that yields a menu descriptor. *MenuItem* is an expression that yields a menu item descriptor.

Examples

```
enable menu "Utilities"
enable menuItem "Get Back" of menu "Direction"
```

Description

The `enable` command enables a specified menu item. When you create a menu item, it is enabled (active) by default. The `enable` command also sets the `enabled` property to `true`.

If you enable a menu that does not exist, HyperCard displays a "No such menu" dialog box. If you enable a menu item that does not exist, HyperCard displays a "No such menu item" dialog box.

Notes

See also the `cmdChar`, `enabled`, `markChar`, and `menuMsg` properties in Chapter 12, "Properties," and the `create menu`, `disable`, and `put` commands in this chapter.

Command keys do not work on menu items that have been deleted or disabled.

EnterKey

Syntax `enterKey`

Description The `enterKey` command sends a statement typed into the Message box to the current card or, if a field is open for text editing, `enterKey` closes the field.

Notes The `enterKey` message, which invokes the `enterKey` command if it reaches HyperCard, is normally sent by pressing the Enter key on the keyboard, but you can also execute it as a line in a script.

Closing a field with `enterKey` doesn't send the `closeField` system message.

See also the `enterKey` message in Chapter 8.

Export paint

Syntax `export paint to file` *fileName*

FileName is an expression that yields any valid Macintosh filename.

Example

```
export paint to file "TreeFrogs"
```

Description The `export paint` command creates a Macintosh paint file containing the image of the current card and saves it with the specified filename. If you are working in the background, only the graphics, buttons, and fields visible in the background are exported. If you are not working in the background, `export paint` exports both the card and background graphics plus all the visible card and background buttons and fields.

Notes If an error is generated while using the `export paint` command, the error is stored in the HyperCard function `the result`.

The `export paint` command only works when a Paint tool is chosen. If you use `export paint` while using the `browse`, `button`, or `field` tool, an error message is put into `the result`.

See also the `import paint` command in this chapter and the `result` function in Chapter 11.

Find

Syntax

```
find text [in field]
find chars text [in field]
find word text [in field]
find whole text [in field]
find string text [in field]
```

Text is an expression that yields a series of one or more text strings separated by spaces, and *field* is an expression that yields a card field or background field descriptor.

Examples

```
find "money" in field "Charity"
find chars "Wild" in field 1
find word msg in second field
```

Description

The `find` command searches through all the card and background fields (visible or not) in the stack for the text strings yielded by *text*. The search begins on the current card and continues through the last card, the first card, and on to the card previous to the current card. Choosing Find from the Edit menu (or pressing Command-F) puts the `find` command in the Message box with the text insertion point after it between double quotation marks.

Script

The following example handler queries the user for search criteria, then executes the `find` command:

```
on doMenu var
  global findString
  if var is "Find..." then
    ask "Find what string:" with findString
    if It is not empty then
      put It into findString
      answer "Match" && findString && "how:" with "Chars" or¬
      "Word" or "All"
      if It is "Chars" then find chars findString
      else if It is "Word" then find word findString
      else find findString
    end if
  else pass doMenu
end doMenu
```

Notes

The `find` command executes faster if you use as many three-character combinations as possible in the search string. That is, three characters are faster than one, six are faster than three, nine are faster than six, and so on.

The `find` form finds the match only at the beginnings of words. The `find chars` form finds the match anywhere within words. The `find word` form matches only complete words.

The `find whole` form (also invoked by pressing Shift-Command-F) lets you search for a specific word or phrase, including spaces. For HyperCard to find a match, all the characters must be in the same field, and they must be in the same consecutive order as they appear in the string derived from *text*.

When you use `find` without `whole`, HyperCard finds a card that contains every word in the string derived from *text*, but the words can appear in different order or in different fields. That is, with `find whole`, interword spaces are part of the search string; without `whole` the spaces delimit separate search strings. With every form of `find`, you can limit the search to a specific background field.

The following example finds a card with a field that has the phrase *Claris Corporation* in it; it won't find *Claris Corp.* or *This corporation is Claris.* (The `find` command without `whole` would find a match in all three cases.)

```
find whole "Claris Corporation"
```

`Find whole` won't find partial-word matches, and it pays no attention to case or diacritical marks: *claris Corporation* and *cLAris cOrporAtion* are seen as the same.

The `find string` form lets you search for a contiguous string of characters, including spaces, regardless of word boundaries. (`Find whole` searches for characters at the beginnings of words.) For HyperCard to find a match, all the characters must be in the same field, and they must be in the same order as in the string derived from *text.* For strings without spaces, `find string` works the same as `find chars`.

In the following example, HyperCard finds the string in *Claris Cororation* but not in *corporation is Claris.* (The `find` command without `string` would not find a match in either case.)

```
find string "Claris Corp"
```

If the match is on a different card, it becomes the current card; if a match isn't found, the current card doesn't change. If you enter the `find` command from the Message box and the match isn't found, HyperCard sounds a beep. If it finds a match, HyperCard puts a box around the word containing the found string if the field containing the string is visible. If a match is found in a hidden field, the field's card becomes the current card, but the field
remains hidden.

As the `find` command evaluates the expression passed to it, it places the resulting values internally between quotation marks as a single parameter string. The following examples show text expressions on the left and the resulting parameter string on the right:

```
find "my" && "word"     find "my word"
find "my" & "word"      find "myword"
find a & b & c          find "xyz"  -- if a = "x",
                                    -- b = "y", c = "z"
find a && b && c        find "x y z"
```

If more than one search string (separated from each other by spaces) is included in the parameter string, all of them must be on a single card or its background for a successful search. However, they can be in any order on the card and only the first is shown with a box around it.

The text in shared background fields and fields with the `dontSearch` property set to `true` is ignored by the `find` command.

Press Command-F to display the parameter string from the most recently executed `find` command in the Message box.

An unsuccessful search sets the HyperTalk function `the result` to `not found`. After a successful search, `the result` is empty. (See the `result` function in Chapter 11.)

For more information about retrieving text, see the `foundChunk`, `foundField`, `foundLine`, and `foundText` functions in Chapter 11.

FunctionKey

Syntax

`functionKey` *keyNumber*

KeyNumber is an expression that yields an integer between 1 and 15.

Examples

```
functionKey 1
functionKey 15
```

Description

The `functionKey` command has built-in Undo, Cut, Copy, and Paste functions for *keyNumber* values 1 through 4, respectively. Any other value of *keyNumber* has no built-in effect.

Script

The following example handler uses the `functionKey` command to implement the message `undo` as a command:

```
on undo
  functionKey 1 -- preprogrammed as undo in HyperCard
end undo
```

Notes

The `functionKey` message, which invokes the `functionKey` command if it reaches HyperCard, is normally generated by pressing one of the 15 function keys on the Apple Extended Keyboard. But you can also send it from the Message box or execute it as a line in a script.

You can program function keys 5 through 15, or reprogram keys 1 through 4, by writing an `on functionKey` handler in the script of any object in the hierarchy between the current card and HyperCard. The following `functionKey` handler opens the Print Report dialog box when function key 9 is pressed:

```
on functionKey whichKey
  if whichKey = 9 then
    doMenu "Print Report..."
    exit functionKey
  end if
  pass functionKey
end functionKey
```

See also the `functionKey` system message in Chapter 8.

Get

Syntax

`get` *expression*

Expression yields any value.

Examples

```
get the long name of field 1
get the location of button "newButton"
get 2+3 -- puts 5 into It
get the date
```

Description

The `get` command puts the value of any expression into the local variable `It`. That is, `get` *expression* is the same as `put` *expression* `into It`.

Script

The following example handler saves the current user level, sets the user level to 5, then restores the saved level:

```
on doMything
   get userLevel -- get the current userLevel
   put It into savedLevel -- save userLevel before changing it
   set userLevel to 5 -- set userLevel for my button or script
   -- (put my script here)
   -- restore userLevel when leaving
   set userLevel to savedLevel
end doMything
```

Go

Syntax

```
go [to] [stack] stackName [in a new window] [without dialog]
go [to] background [of [stack] stackName] [in a new window]¬
[without dialog]
go [to] card [of background] [of [stack] stackName]¬
[in a new window] [without dialog]
```

StackName is an expression that yields a valid stack name. *Card* is an expression that yields a valid card descriptor. *Background* is an expression that yields a valid background descriptor.

Examples

```
go card 23
go to stack "ArtIdeas"
go bkgnd field 1 -- if bkgnd field 1 contains a stack name
go "home"
go mid card of stack "clip art"
go next
go to first card of second background of "home"
go card 2 of stack "VacationSpots" in a new window
go stack "VacationSpots" in a new window without dialog
go card 4 of stack "VacationSpots" without dialog
go "hd:bigFolder:innerFolder:myStack" -- full pathname
```

Description

The `go` command takes you to the specified destination. If you name a stack without specifying a card, you go to the first card in the specified stack. If you don't name a stack, you go to the specified card in the current stack. If you go to a background, you go to the next card with that background (not the first card). If the current card has the specified background, you won't move. You can specify a visual effect to be used on opening the card by issuing the `visual effect` command before you use the `go` command.

If the destination is in a stack other than the current one and you use the `in a new window` form of the `go` command, the destination stack is opened in addition to any existing stacks. If you do not specify the `in a new window` form, the current stack is closed before the specified stack is opened. If only one stack is open and you do not specify the `in a new window` form of the go command, the current stack closes and the specified stack appears without a close box.

If you use the `without dialog` form of the `go` command and the destination can't be found, you do not get a standard dialog box for opening files. Instead, `the result` is set to "No such stack" or "No such card."

Script

The following example handler queries the user for a destination, then executes a `go` command with a visual effect:

```
on mouseUp
  ask "Where to?" with "This card"
  if It is empty then put "this card" into It
  put It into goWhere
  visual effect fade to black
  go to goWhere
end mouseUp
```

Help

Syntax

```
help
```

Description

The `help` command takes you to the first card of the stack named HyperCard Help.

Note

See also the `help` system message in Chapter 8.

Hide

Syntax

```
hide menuBar
hide object
hide groups
hide card picture
hide picture of card
hide background picture
hide picture of background
hide titlebar
hide window windowName
```

Object yields one of the following:
a valid button descriptor in the current stack
a valid field descriptor in the current stack
`message [box] or message [window]`
`window "navigator"` (the navigator palette)
`pattern window` or `window "patterns"` (the Patterns palette)
`tool window` or `window "tools"` (the Tools palette)
`message watcher` or `window "message watcher"`
`variable watcher` or `window "variable watcher"`
`card window`

Card yields the descriptor of a card in the current stack. *Background* yields the descriptor of a background in the current stack. *WindowName* is the name of a window.

Examples

```
hide message
hide bkgnd button "goHome"
hide field id 1
hide window "Planets"
hide card window
```

Description

The `hide` command removes the specified object from view. Its effect is the same as setting the `visible` property of the specified object to `false`, or clicking a window's close box.

The `hide picture` form of the `hide` command removes from view the graphic bitmap on the card or background, and the `show picture` form of the `show` command displays it.

The `hide groups` form of the `hide` command removes the gray two-pixel underline made visible below group style text by the `show groups` form of the `show` command. The `hide groups` command affects all group style text in all fields globally.

Script

The following example handler hides a field or button when the user puts the pointer over the button or field:

```
on mouseWithin
   hide the target
end mouseWithin
```

Notes

`Message` can be abbreviated `msg`. `Background` can be abbreviated `bkgnd`. `Button` can be abbreviated `btn`. `Card` can be abbreviated `cd`.

`Hide menuBar` is also sent as a system message when the menu bar is hidden.

If the screen is locked, `hide menuBar` has no effect. See the `lock screen` and `unlock screen` commands later in this chapter and the `lockScreen` property in Chapter 12, "Properties."

The `hide` command does not affect the `location` property of an object or window. You can't use *the* with the object name. For example, the following statement results in the error message "Can't understand arguments of command hide."

```
hide the window "navigator"
```

Hidden fields aren't in the tab order. (They are skipped when you move the text insertion cursor from one visible field to the next by pressing the Tab key.) The `find` command does search through them, however, and you can put values into them and put their values elsewhere.

Hidden card and background pictures are not displayed when the Browse, Button, or Field tools are chosen, but if you attempt to use a Paint tool manually, a dialog box appears asking if you want to make the picture visible; clicking OK displays the picture. (You can draw on hidden pictures from a script.) Whether or not you are working in the background determines whether your actions pertain to the card or background picture.

If there is another card window behind the current stack's card window and you hide the current stack's card window, the card window behind it becomes the current card window.

If you use either the `hide card window` or `hide window` *stackName* form of the `hide` command to hide the current card or specified stack, the cards in that stack won't be visible again until you set the card window or stack window's `visible` property to true, use the `show card window` or `show window` *stackName* command, or close the stack and reopen it.

You can move hidden windows without changing their visible state with the `location` property, which is described in Chapter 12, "Properties."

See also the `picture` and `show` commands, later in this chapter, and the `visible` property in Chapter 12, "Properties."

Import paint

Syntax

`import paint from file` *fileName*

FileName is an expression that yields a valid Macintosh filename.

Description

The `import paint` command reads in the specified paint file and makes it the current selection. The `import paint` command allows you to put digitized images and other pictures created with Macintosh paint programs onto backgrounds or cards.

You can use the `export paint` and `import paint` commands together to enhance graphics created with HyperCard. For example, you can export a card image to a file and open it in MacPaint or your favorite paint application. You can then paste in your own pictures and make changes to the file containing the HyperCard card art with your paint program. Once you complete the art work, import the finished paint file to the original card or new card or background.

Note

If an error is generated while using the `import paint` command, the error is stored in the HyperCard function `the result`.

The `export paint` command only works when a Paint tool is chosen. If you use `export paint` while using the `browse`, `button`, or `field` tool, an error message is put into `the result`.

Imported pictures are clipped to the size of the current card. The maximum area of an imported paint file that can be displayed is 576 pixels by 720 pixels.

See also the `export paint` command in this chapter and the `result` function in Chapter 11.

Lock screen

Syntax `lock screen`

Example `lock screen`

Description The `lock screen` command sets the `lockScreen` global property to `true`, preventing HyperCard from updating the screen. If you go to another card or do other actions that change the appearance of the screen, those changes are not displayed until the `lockScreen` property becomes `false`.

Note See also the `unlock screen` command in this chapter and the `lockScreen` property in Chapter 12.

Mark

Syntax

```
mark card
mark cards where expression
mark all cards
mark cards by finding text [in field]
mark cards by finding chars text [in field]
mark cards by finding string text [in field]
mark cards by finding whole text [in field]
mark cards by finding word text [in field]
```

Card is an expression that yields a card descriptor. *Expression* is an expression that yields the criteria on which you want to base the marking of cards. *Text* is an expression that yields the text criteria on which you want to base the marking of cards. *Field* is an expression that yields a field descriptor.

Examples

```
mark [the] next card
mark cards where "We be shaking" is in field 2
mark all cards
mark cards by finding whole chicken in field 1
```

Description

The `mark` command sets the `marked` property for the specified card or cards to `true`. Cards can also be marked with the Card Marked option in the Card Info dialog box. By default, cards are unmarked.

The `by finding` form of the `mark` command uses `chars`, `word`, `whole`, and `string` to define the search criteria the same way the `find` command does. See the description of the `find` command for information about how to use these forms.

The `mark` command can be used with the `unmark` command in searches where you want to find and mark cards containing particular information while excluding those with other information.

For example, say you want to mark and print all cards with information about "San Francisco" but not "earthquake." You might have a script that uses the following statements:

```
unmark all cards
mark cards where "San Francisco" is in field 1
unmark cards where "earthquake" is in field 1
```

You could then show, print, or display the number of cards containing that combination of search criteria.

```
show marked cards
print marked cards
put the number of marked cards into word 3 of field 2
```

Note

See also the `marked` property in Chapter 12, "Properties," and the `unmark` command, later in this chapter.

You can't mark cards in a read-only stack, even if the `userModify` property is set to `true`. See the `userModify` property in Chapter 12, "Properties."

Multiply

Syntax

`multiply [`*chunk* `of]` *container* `by` *number*

Chunk is an expression that yields a chunk expression. *Container* is a container holding a numeric value and *number* is an expression that yields a numeric value.

Examples

```
multiply Subtotal by Tax
multiply field 1 by field 3
multiply line 3 of card field 2 by 25
multiply It by 2 -- puts result into It, replacing the old value
```

Description

The `multiply` command multiplies the value in `[chunk of]` *container* by the value of *number* and puts the result in *container*.

Script

The following example handler adds 6 percent to the value of items in a list:

```
on taxMe
  put "12.45,15.00,150.00,76.95,10.00,14.95" into taxables
  repeat with count = 1 to the number of items in taxables
    multiply item count of taxables by 1.06
  end repeat -- the new values are stored in taxables
end taxMe
```

Notes

The value previously in the container must be a number; it is replaced with the new value.

The result is calculated to a precision of up to 19 decimal places and, if put into a field or the Message box, is displayed according to the `numberFormat` global property.

See also the `numberFormat` global property in Chapter 12 and the discussion of numbers in Chapter 6.

Open

Syntax

`open [`*fileName* `with] `*application*

Application is the name of any application and *fileName* is the name of any document on your Macintosh. Either one can be an expression that yields such a name.

Examples

```
open "Apps:BigApp"
open "Letter" with "Apps:MacWrite"
open Field 3
open FavoriteApp
```

Description

The `open` command launches the named application. A specific document may be opened with its own creator or a compatible application by using the `open` *filename* `with` *application* form of the `open` command.

Script

The following example handler queries the user for a document and application before executing the `open` command:

```
on mouseUp
  ask "Open what document?" with empty
  if It is not empty then
    put It into doc
    ask "Use what application?" with empty
    if It is not empty then open doc with It
  end if
end mouseUp
```

Notes

If the document or application you specify isn't at the top level of the file hierarchy (the "disk" level), then the path to it must be specified on the appropriate Search Path card of the Home stack. Alternatively, you can specify the full pathname with the `open` command:

```
open "MyHardDisk:Apps:Apples"
```

If HyperCard can't find the requested document or application, it displays the dialog box for locating files to the user. If HyperCard can't open the application under MultiFinder because of low memory, it puts the error in `the result`.

When you quit the application, you go to the card you were on in HyperCard when you executed the `open` command. However, any global variables you had previously declared are now gone, and any portions of handlers that remained unfinished when you executed the `open` command do not finish.

Open file

Syntax

```
open file fileName
```

FileName is the name of any file on your Macintosh, or an expression that yields such a name.

Examples

```
open file "textOnly"
open file field 1
```

Description

The `open file` command opens the named file for reading or writing. Usually, the file is an ASCII text file opened in preparation for importing or exporting text. If the specified file doesn't exist, HyperCard creates it.

Script

The following example handler determines if a given file exists by trying to read from it:

```
on checkFile
  put "MyFilename" into filename
  open file filename
  read from file filename for 16384
  if It is empty then answer "File does not exist" with "OK"
  close file filename
end checkFile
```

Notes

If the specified file is already open, an error is generated. The error is stored in the HyperCard function `the result`. `The result` is empty if the command is successful. Use the `close file` command to close files explicitly after you use them. HyperCard automatically closes all open files when an `exit to HyperCard` statement is executed, when you press Command-period, or when you quit HyperCard.

You must provide the full pathname of the file if it's not at the same directory level as HyperCard. (See "Identifying a Stack" in Chapter 5 for an explanation of pathnames.)

See also the `read`, `write`, and `close file` commands, in this chapter and the `result` function in Chapter 11.

Open printing

Syntax

```
open printing [with dialog]
```

Description

The `open printing` command starts a print job to be ended later by a `close printing` command.

The settings specified in the Print Stack dialog box are used unless `with dialog` is specified, in which case the dialog box is displayed and new settings can be chosen.

Script

The following example handler prints a selection of cards:

```
on printSelection
  put "1,3,8,15,21" into myCards
  open printing with dialog
  repeat with count = 1 to the number of items in myCards
    go card item count of myCards
    print this card
  end repeat
  close printing -- print the cards
end printSelection
```

Notes

Printing cards with `open printing` is similar to printing with the Print Stack command in the File menu, except that Print Stack prints all cards in the stack, while `open printing` prints only the ones you specify with the `print card` command, described later in this chapter.

You must use some form of the `print` command and then the `close printing` command to print and then close a print job begun with `open printing`. Don't use the `print` *fileName* `with` *application* command while a print job is active.

See also the `close printing`, `open report printing`, `print`, and `print card` commands in this chapter.

Open report printing

Syntax

```
open report printing [with template templateName]
open report printing [with dialog]
```

TemplateName is an expression that yields the name of a previously defined print template.

Examples

```
open report printing
open report printing with template "fieldsOnly"
open report printing with dialog
```

Description

The `open report printing` command prepares a report printing job. The print report job is sent to the printer and closed with a `close printing` command.

Script

The following script sets the print template and prints the marked cards in the current stack.

```
on PrintLabels
  open report printing with template mailing labels
  -- choose the mailing labels template
  print marked cards -- specify which cards to print
  close printing -- generate the report & print
end PrintLabels
```

Notes

Settings previously specified in the Print Report dialog box are used unless the `with dialog` form is used, in which case the Print Report dialog box is displayed and new settings can be chosen.

If you do not specify a print report template with the `with template` *templateName* form or do not use the `with dialog` form, the settings from the last print report template used for a print job are used for the current print job. If no print report template has been used previously for printing and you neither specify a print report template nor use the `with dialog` form, nothing is printed.

Printing cards with `open report printing` is similar to printing with the Print Report command in the File menu, except that you can specify the report template to use in the script without using the Print Report dialog box.

You must use some form of the `print` command and the `close printing` command to send the job to the printer and then close a print job begun with `open report printing`. If you start another print job without closing the previous one, HyperCard notifies you that it will close the previous print job before starting the new one. Don't use the `print` *fileName* `with` *application* command while a print job is active.

When you choose cancel after opening the Print Report dialog box with `open report printing`, the function `the result` is set to `cancel`.

See also the `close printing`, `open printing`, `print`, and `print card` commands in this chapter.

Palette

Syntax

`palette` *paletteName* `[,` *point* `]`

PaletteName is an expression that yields the name of the palette you wish to invoke. *Point* is an expression that yields two comma-separated integers that represent the horizontal and vertical coordinate at which the palette should appear. The point is the offset from the upper-right corner of the current card window to the upper-right corner of the palette minus the title bar. *Point* needs to be in quotation marks or passed in a container.

Examples

```
palette "navigator", "50,100"
palette "GeneralPalette", "150,80"
```

Description

The `palette` command displays the specified custom XCMD palette or the HyperCard palette called navigator. If you do not specify a point parameter, the palette appears at the default location of 10,20 inside the current card coordinate system if it's the first time it's been displayed. After the first time, it appears at its last location if you don't specify a point. The point is reset to the default location at the beginning of each HyperCard session.

The `palette` command and associated palette properties have no effect on the Tools palette or Patterns palette.

Notes

If you specify a palette that is already visible, the `palette` command moves the palette to the location specified by the point parameter.

You can set the `location` property of a palette window after the palette is displayed with the `palette` command:

```
set the loc of window "navigator" to "65,80"
```

You can close a palette that is displayed by using the `close` command:

```
close window "navigator"
```

After a palette has been displayed with the `palette` command, you can hide and show a palette by using the `hide` and `show` commands or setting the `visible` property. The `hide` and `show` commands have no effect if the palette hasn't been displayed yet by the `palette` command. Here are some examples of statements that hide or show palettes:

```
hide window "navigator"
set the visible of window "navigator" to true
show window "GeneralPalette"
```

There is a group of properties that apply only to HyperCard XCMD palettes. The palette properties are `buttonCount`, `commands`, `hilitedButton`, and `properties`.

You can use the `buttonCount` property to determine the total number of buttons in a palette.

`Commands` returns a return-delimited list of the commands or messages associated with the palette's buttons. The commands are listed according to the number of the button they are associated with, that is, first the command associated with button number 1, then the command associated with button number 2, and so forth.

`HilitedButton` determines or sets the number of the currently highlighted button of the specified palette. Setting `HilitedButton` to `true` does not cause the message associated with the button to be sent. For example, the statement

```
set the hilitedButton of window "navigator" to 3
```

does not send the `doMenu "Help"` message associated with button 3 of the navigator palette.

In the case of action palettes, such as navigator, the value of the `hilitedButton` property is always `-1`.

`Properties` returns a comma-separated list of the names of the properties that apply to the specified palette.

HyperCard palettes consist of two resources: one of type 'PLTE' and one of type 'PICT'. The PLTE resource contains the configuration information used by the palette XCMD, and the PICT resource supplies the palette image. The two resources must have the same name and resource ID number. Because the palette image is a PICT resource, palettes can be in color.

See also the `hide` and `show` commands in this chapter, and see Appendix A for more information about the XCMD callbacks for palettes.

Picture

Syntax

`picture` [*fileName, fileType, windowStyle, visible,depth*]

FileName is the name of a file or resource of type 'PICT' or 'PNTG' on your Macintosh, or an expression that yields such a name.

FileType is `resource`, `file`, or `clipboard`. (The default file type is `file`.)

WindowStyle is the style of window in which the picture is displayed. The window styles are `plain`, `rect`, `zoom`, `roundRect`, `dialog`, `document`, `shadow`, and `windoid`. (The default window type is `zoom`.)

Visible is a boolean value; `true` for visible, `false` for invisible. This parameter allows you to create an invisible window and set its properties before displaying it with a `show` command. See the description for more information about the picture window properties.

Depth is the bit depth of the off-screen buffer that the `picture` command creates. Bit-depth values between `0` and `32` inclusive are supported. The `picture` command allocates an off-screen buffer of the bit depth you specify (rounded down to a power of 2) or the bit depth of the picture file, whichever is smaller. This allows you to display picture files with a deeper bit depth in less memory, but at the cost of lower resolution. If the value is `0`, the `picture` command doesn't create any off-screen buffer, instead the PICT file is drawn directly into the window. A value of `0` for depth makes scrolling and zooming slower and prevents dithering from working. If available system memory is extremely low and the bit depth is set to `0`, the file is spooled to the display, but `picture` properties do not work. It does, however allow you to display pictures in less memory and display animated PICT files and format 1 PICT files with color.

Examples

```
picture "Clowns",resource,plain,false,0
picture "MyPICT",file,rect
picture "Picnic",clipboard,roundRect
```

Description

The `picture` command displays color or gray-scale pictures in an external window. The pictures can come from the Clipboard, from PICT or MacPaint® files, or from PICT resources in your stack.

The `picture` command works best if the HyperCard application memory size in the Get Info dialog box is set to 2 megabytes or more. If the picture cannot be displayed because of insufficient memory, an error describing the condition is returned in the `result` function.

There is a set of properties that apply to windows created with the `picture` command. They are `rect`, `globalRect`, `loc`, `globalLoc`, `scroll`, `zoom`, `scale`, and `dithering`. The properties are in addition to the standard HyperTalk properties `location` and `visible`, which also apply to windows created with the `picture` command.

`Rect` or `globalRect` is specified as four comma-separated integers representing the bounding window rectangle. The first two integers represent the upper-left corner position of the window on the screen and the second two integers represent the lower-right corner of the window. Four literals can also be used to set the `rect` and `globalRect` property. They are `cardScreen` (or `card`), `largestScreen` (or `largest`), `deepestScreen` (or `deepest`), and `mainScreen` (or `main`). These literals display the picture window centered on the same screen as the card window, on the screen with the largest area, on the screen with the greatest bit depth, and on the main screen, respectively. Here is an example that sets the `rect` and `globalRect` properties:

```
set the rect of window "Clowns" to "120,225,300,480"
set the globalRect of window "Clowns" to "largest"
```

The `loc` and `globalLoc` properties are set just like the `location` property for other HyperCard windows. The `loc` property applies to the location of the window created with the `picture` command in coordinates local to the current card window. `GlobalLoc` applies to the location of the window created with the `picture` command in global screen coordinates. See the `location` property in Chapter 12, "Properties."

`Loc` or `globalLoc` specify a point in the screen's coordinate system where the upper-left corner of the window is to be displayed. The point is specified as two integers that represent the horizontal and vertical offset, respectively. Four literals can also be used to set the `loc` and `globalLoc` property. They are `cardScreen` (or `card`), `largestScreen` (or `largest`), `deepestScreen` (or `deepest`), and `mainScreen` (or `main`). These literals display the picture window centered on the same screen as the card window, on the screen with the largest area, on the screen with the greatest bit depth, and on the main screen, respectively. Here are examples that set the `loc` and `globalLoc` properties:

```
set the loc of window "Flowers" to "65,100"
set the globalLoc of window "Clowns" to "card"
```

The `scroll` property for windows created with the `picture` command is like the `scroll` property for cards in card windows. See the `scroll` property in Chapter 12, "Properties." `Scroll` is specified as two comma-separated integers representing the horizontal and vertical offset, in the picture's coordinate system, to be displayed at the upper-left corner of the window. Here is an example that sets the `scroll` property for a window.

```
set the scroll of window "Water" to "45,60"
```

The `dithering` property is a boolean value; `true` for dithering, `false` for no dithering. (The default for the `dithering` property is `false`.) Here is an example that sets the `dithering` property for a window.

```
set the dithering of window "Garlic" to true
```

The `scale` property scales a picture in a window created with the `picture` command. The value for `scale` is an integer between `-5` and `5`, inclusive. Negative integers scale down the picture and positive integers scale up the picture. Scaling is done using the log base 2 of the specified `scale` value. The default value for `scale` is 0. Here is an example that sets the `scale` property for a picture in a window.

```
set the scale of window "Summer fun" to 4
```

The `zoom` property applies to windows created with a window style that supports a zoom box. It zooms a window in or out. The possible values for `zoom` are `in` or `out`. Here is an example that sets the `zoom` property for a window.

```
set the zoom of window "Alligators" to out
```

Notes

If you do not provide a valid name for the *fileName* parameter, a standard dialog box for opening files is displayed from which you can choose a PICT or PTNG file. The one exception to this is when you specify `clipboard` for the *fileType* parameter. In that case any name can be used. If you cancel the standard file dialog box, `picture` sets `the result` to `cancel`.

If the *fileType* parameter is `resource` and the filename specified in the *fileName* parameter can't be found, the `picture` command converts the *fileName* parameter to a number and looks for a PICT resource with that number as its ID.

If you do not set the `rect` or `globalRect` property, HyperCard displays the picture in a window that is the same size, or as close to the same size as possible, as the original picture.

If you do not set the `scroll` property, the picture is displayed with its 0,0 coordinate at the upper-left corner of the external window.

The new 32-bit Color QuickDraw is fully supported. You can display 16-, 24-, and 32-bit images with the `picture` command. The `dithering` property is ignored if 32-bit color QuickDraw isn't installed in the System Folder.

The `rect` and `shadow` window types are created behind all the HyperCard windows. This allows you to create picture windows that appear to pop up above the card window.

You can close windows created with the `picture` command by clicking the close box or with the `close` command as follows:

```
close window "fileName"
```

FileName is the filename specified in the *fileName* parameter for the `picture` command. It is also the name displayed in the title bar of the window.

If an error occurs when creating a window, the `picture` command sets `the result` to an error message that begins with "Couldn't display picture."

If the picture file is displayed successfully, `the result` is empty.

When you click in a window created with the `picture` command, two system message are sent. When the mouse is down a `mouseDownInPicture` is sent. When the mouse button is released, a `mouseUpInPicture` is sent. Each message is sent with two parameters: the name of the window and the point within the picture's default (not scaled) local coordinates at which the mouse was clicked. You can place handlers for these messages anywhere in the message-passing hierarchy or in the stack script of the stack that invokes the `picture` command. You can use these messages to simulate button actions, card flipping, or anything else that you would use `mouseUp` and `mouseDown` system messages for.

The following handler checks for a `mouseDownInPicture` message sent by a window created by the `picture` command and puts its name and the location where the mouse was clicked into the Message box.

```
on mouseDownInPicture wName,cLoc
   put "You clicked window" &&quote& wName &quote&& "at ¬
   location" && cLoc
end mouseDownInPicture
```

See also the `close` command in this chapter, and the system messages `mouseUpInWindow`, `mouseDownInWindow`, `openPicture`, and `closePicture` in Chapter 8, "System Messages."

Play

Syntax

```
play sound [tempo] [notes]
play stop
```

Sound is an expression that yields the name of a digitized sound. (`Boing` and `harpsichord` are included with HyperCard.) *Tempo* is an expression that yields the speed at which the sound plays, and *notes* is an expression that yields a list of one or more notes representing the pitch at which the sound plays and the duration of the notes. Quotation marks around *sound* and *notes* are required. Digitized sounds are of the Macintosh sound resource formats 1 and 2, which are described in *Inside Macintosh.*

Examples

```
play "boing"
play "harpsichord" "d"
```

Description

The `play` command makes the Macintosh play notes through its speaker (or through the audio jack if it's plugged in). The `play stop` form stops the current sound immediately; otherwise it plays until it's done and stops by itself. HyperCard continues to execute handlers and perform other actions while a sound plays.

Script

The following example handler goes to each card in a stack and synchronizes playing the specified note with each card change:

```
on tour
  repeat the number of cards
    play "harpsichord" tempo 200
    go next card
    wait until the sound is "done"
  end repeat
end tour
```

Notes

The *tempo* is a number specifying the speed at which the note is played (100 is a medium tempo; higher numbers are faster). The sound and tempo are specified once for each `play` command.

Pop card

Syntax

`pop card [`*preposition* `[`*chunk* `of] `*container*`]`

Preposition is `into`, `before`, or `after`, *chunk* is a chunk expression, and *container* is an expression that yields a container.

Example

```
pop card into field 3 of card WhereIbeen
```

Description

The `pop card` command retrieves the identification (full ID and stack pathname) of a card previously saved with the `push card` command. If you don't provide a destination for the identification, you go directly to the card whose address is popped.

Script

The following example handler pushes whatever card you're on, goes to another stack, gets the value of a field property, then returns to the original card:

```
on getTheFont
   global myStack,theFont
   push card
   go myStack
   put textFont of field 1 into theFont
   pop card -- goes to the card formerly pushed
end getTheFont
```

Notes

If you don't specify a container, after the card has been popped, its identification is removed from the memory stack—it can't be popped again. If a container is given, however, the card's identification is put into the container, but you don't go anywhere.

See also the `push` command, later in this chapter.

Print

Syntax

```
print fileName with application
print field
print expression
```

FileName is an expression that yields the name of any document on your Macintosh, and *application* is an expression that yields the name of the application to which it belongs (or with which it is compatible). *Field* is an expression that yields any field descriptor. *Expression* yields an arbitrary expression or container.

Examples

```
print "memo" with "MacWrite"
print field 1 with field "Program"
print "HD:MY DOCS:letter" with "HD:Applications:MacWrite"
print bkgnd field 1
```

Description

The `print` command prints the specified file, field, or expression.

The `print` *fileName* with *application* form of the `print` command suspends HyperCard, launches the named application, opens the named document, prints the document, and then resumes running HyperCard. The specified application must support printing.

The `print` *field* form of the `print` command prints the specified field using the current font, size, style and line height of that field.

The `print` *expression* form prints any arbitrary HyperTalk expression or container.

Script

The following example handler queries the user for the name of a document to print and an application with which to print it:

```
on mouseUp
  ask "Print what document?" with empty
  if It is not empty then
    put It into doc
    ask "Use what application?" with empty
    if It is not empty then print doc with It
  end if
end mouseUp
```

Notes

If the document or application you specify isn't at the top level of the file hierarchy (the "disk" level), then the path to it must be specified on the appropriate Search Path card of the Home stack. Alternatively, you can specify the full pathname with the `print` command.

Don't use the `print` command while a print job started with the `open printing` command is active.

See also the `print card` command described in this chapter.

Print card

Syntax

```
print card [from point1 to point2]
print number cards
print marked cards
print card [from point1 to point2]
```

Point1 is an expression that yields two comma-separated numbers representing the upper-left corner of a rectangular area of the current card that you want to print. *Point2* is an expression that yields two comma-separated numbers representing the lower-right corner of a rectangular area of the current card that you want to print. *Number* is an expression that yields an integer or the word `all`, and *card* is an expression that yields the card descriptor of a card in the current stack.

Examples

```
print card from 0,123 to 345,512
print last card
print card id 3011
print all cards
print marked cards
print howMany cards -- howMany contains a number or "all"
print card
```

Description

The `print card` command makes HyperCard print the current card, the same as the Print Card command (Command-P) in the File menu. The `print` *number* `cards` form prints the number of cards specified by *number*, or all the cards in the stack, beginning with the current card. The `print marked cards` form prints a group of marked cards. The `print` *card* form makes HyperCard go to the specified card, print it, and return to the current card.

Script

The following example handler queries the user for the number of cards to print whenever Print Card is chosen from the File menu:

```
on doMenu var
   if var is "print card" then
      ask "Print how many cards?" with one
      open printing
      print It cards
      close printing
   else pass doMenu -- make sure other menu choices
   -- continue to work
end doMenu
```

Notes

You don't need to use the `open printing` command before using the `print card` command. If nothing is printing, the `print card` command prints the specified card or cards immediately; if an `open printing` command is in effect, no cards are printed until a page is full (depending on how many cards per page are specified in the printing dialog box) or the `close printing` command is given.

Chapter 5, "Referring to Objects, Menus, and Windows," defines card descriptors. See also the `marked` property in Chapter 12, "Properties," and the `close printing`, `mark`, `open printing`, and `open report printing` commands described earlier in this chapter.

Push

Syntax

```
push card
```

Card is an expression that yields the descriptor of any card in the current stack.

Examples

```
push recent card
push first card
push card
```

Description

The `push` command saves the identification of the specified card in a LIFO (last-in, first-out) memory stack (an area of memory, not a HyperCard stack).

Script

The following example handler saves the current card, goes to a random card, then returns to the original card:

```
on nonSense
  push card -- save current card
  go any card
  pop card -- restore current card
end nonSense
```

Notes

The card identification can be retrieved later with the `pop card` command (usually so that you can go directly back to the pushed card). The card identification that's saved is the full card ID and stack pathname. HyperCard holds the IDs of up to 20 cards.

Card descriptors are described in Chapter 5, "Referring to Objects, Menus, and Windows."

See also the `pop card` command, earlier in this chapter.

Put

Syntax

`put` *expression* [*preposition* [*chunk* `of`] *container*]

`put` *itemName preposition* [*menuItem* `of`] *menu* [`with menuMsg` *message*]

Expression is an expression that yields a text string or number. *preposition* is `into`, `before`, or `after`, *chunk* is a chunk expression. and *container* is an expression that yields a container.

ItemName is an expression that yields a single menu item name or a list of comma-separated or return-delimited names to be added to the specified menu. *MenuItem* is an expression that yields the word `menuItem` followed by either the name or number (integer or ordinal number) of a standard HyperCard menu item or a user-defined menu item in the specified menu. *Menu* is an expression that yields the word `menu` followed by either the name or number (integer or ordinal number) of a standard HyperCard menu or a user-defined menu in the menu bar. *Message* is an expression that yields a single message or a list of comma-separated or return-delimited menu messages to be sent when a specified menu item is chosen. The menu messages in the list correspond one to one for each menu item.

Examples

```
put "Hello" into field 1
put "go " before field "WhereTo"
put empty into It
put It -- puts contents of It into Msg
put "Tom" into first word of field "Name"
put "." after first character of last word of field 3
put fld 2 + fld 3 into fld 4 -- adds numbers in fields
put the date into varName
put "Paths" after menuItem "Preferences" of menu "Home"
put "Paths" after first menuItem of menu "Home" with menuMsg¬
"go card 4"
put "Vanilla,Chocolate,Strawberry" into menu "Flavors" with menuMsg¬
"put Yummy,put Tastie,put BerryGood"
```

Description

The first form of the `put` command causes HyperCard to evaluate *expression* and copy the result into *container*. You use the second form of the `put` command to add menu items to an existing menu. Optionally, you can specify a message to be sent when the menu item it belongs to is chosen.

User-defined items with the same name as a standard HyperCard menu items inherit the standard behavior of the HyperCard menu item. For example, if you put an item called Background into a menu called Special, choosing it has the same effect as the standard Background menu item from the Edit menu unless you assign a custom menu message or intercept the `doMenu` message.

HyperCard does not automatically check, uncheck, enable, or disable user-defined menu items as it does for its own standard menu items. It is your responsibility to make sure user-defined menu items act according to the standard Macintosh user interface. See the commands and properties listed in the notes section for more information about controlling the behavior of menu items.

Script

The following example handler initializes three global variables when the stack it's in is opened:

```
on openStack
  global var1,var2,var3
  put 0 into var1
  put empty into var2
  put empty into var3
end openStack
```

Notes

If you don't specify the destination container, the value is copied into the Message box. (HyperCard shows the Message box if it's hidden.) If you specify a container that HyperCard doesn't recognize, it creates a new local variable of that name and puts the value into the variable.

For the `put` *expression* form, using `into` replaces the contents of the container, `before` places the source value at the beginning of the previous contents, and `after` appends the source value to the end of the previous contents.

If *expression* is a container holding an arithmetic expression, the expression is not evaluated but is copied literally into the destination. Use the `value` function with the container name to have HyperCard evaluate its contents.

You can delete the contents of a container by putting the constant `empty` or `""` into it (but this doesn't delete the container). You can specify a chunk expression to insert, replace, or delete a portion of the contents.

Always use the form `put` *itemName* `before|after` *menuItem* `of` *menu* to add a menu item to a menu that already has menu items in it. If you use the form `put` *menuItem* `into` *menu* you replace the contents of the menu, deleting any other menu items already in the menu.

Because menus are like containers, you can get a list of the current menu items in a menu by using the term `menu` as an expression. For example, the following statement puts a return-delimited list of all the menu items in the Home menu into card field "MenuItemList."

```
put menu "Home" into card field "MenuItemList"
```

You can get a specified menu item name with a statement like

```
put menuItem 3 of menu "Home"
```

The maximum number of menu items in a menu is 64.

See also the `checkMark`, `cmdChar`, `enabled`, `menuMsg`, `name`, and `textStyle` properties Chapter 12, "Properties." See also the `create menu`, `delete`, `disable`, and `enable` commands in this chapter.

Read

Syntax

```
read from file fileName at start for numberOfChars
read from file fileName until char
read from file fileName for numberOfChars
```

FileName is an expression yielding the name of any file on your Macintosh; *char* is an expression yielding a character; *start* is an expression that yields a number that indicates the starting point (offset from the beginning of the file) from which reading occurs; and *numberOfChars* is an expression yielding an integer representing the total number of characters to read.

Examples

```
read from file "import" at 4 for 20
read from file "import" until tab
read from file "File Names" until return -- reads one line
read from file "someText" for 16384 -- maximum block size
```

Description

The `read` command reads ASCII characters from the specified file, previously opened with the `open file` command, into the local variable `It`. Reading starts at the position specified with `at` *start* in a newly opened file and continues for the number specified in *numberOfChars*. If no start is specified, then it reads from the beginning of the file or from the last point read with a read command until the character or number of characters is reached.

The `until` *character* form causes reading to stop when the specified character has been read; the `for` *numberOfChars* form causes reading to stop when the specified number of characters (or bytes) have been read. Return characters at the end of lines count, as do space and tab characters.

Script

The following example handler opens a file, reads to the end of the file while placing its contents into a global variable, and closes the file:

```
on mouseUp
  global fileName, textHolder
  open file fileName
  repeat
    read from file fileName for 16834
    if It is empty then exit repeat
    put It after textHolder
  end repeat
  close file fileName
end mouseUp
```

Notes

You can read up to 16,384 characters at a time. If you try to read more characters than that, all but the last 16,384 that you read are ignored. The `read` command puts the characters into the local variable `It`, replacing its previous contents. So, you must put each block of text that you read into another container. (Use `after` with the `put` command to append each new block of text to the end of the previous contents.) Containers other than fields have no practical size limit (they're limited by available memory). If you try to put more than 30,000 characters into a field, the extra characters are ignored.

- *Tab characters:* HyperCard reads tab characters from a file into `It`, and the tab characters remain when you put the text into another variable or a field (where they are displayed as spaces). If you alter any text in the field, however, HyperCard removes the tab characters.

If you specify more than one character with the `until` *character* form, HyperCard stops reading when it matches the first character specified.

Use the `close file` command to close files explicitly after you use them. HyperCard automatically closes all open files when an `exit to HyperCard` statement is executed, when you press Command-period, or when you quit HyperCard.

You must provide the full pathname of the file if it's not at the same directory level as HyperCard. (See "Identifying a Stack" in Chapter 5 for an explanation of pathnames.)

If an error is generated while using the `read` command, the error is stored in the HyperCard function `the result`.

See also the `close file`, `open file`, and `write` commands in this chapter, and the `result` function in Chapter 11.

Reset menuBar

Syntax

```
reset menuBar
```

Description

The `reset menubar` command reinstates the default HyperCard menus and removes any custom menus created with the `create` command and custom menu items put into standard HyperCard menus with the `put` command.

Notes

If you are creating a stack to be used by others, use this command with some restraint because it removes all custom menus and menu items, not just those you created for your stack.

When your stack is closed, remove any custom menus or menu items you created by deleting them. Use a `closeStack`, `suspend`, or `suspendStack` system message handler to remove your custom menus when your stack is closed or is no longer the current stack, and an `openStack`, `resume`, or `resumeStack` system message handler to reinstate your custom menus when your stack is opened or resumed.

See also the `create menu` and `put` commands, earlier in this chapter, and Chapter 8, "System Messages."

Reset paint

Syntax `reset paint`

Description The `reset paint` command reinstates the default values of all the painting properties. The painting properties and their default values are

```
grid          false
lineSize      1
filled        false
centered      false
multiple      false
multiSpace    1
pattern       12
brush         8
polySides     4
textAlign     left
textFont      geneva
textSize      12
textStyle     plain
textHeight    16
```

Note The painting properties are described in Chapter 12, "Properties."

Reset printing

Syntax `reset printing`

Description The `reset printing` command reinstates the default values of all the printing properties. The printing properties and their default values are

```
printMargins     0,0,0,0
printTextAlign   left
printTextFont    Geneva
printTextHeight  13
printTextSize    10
```

Note The printing properties are described in Chapter 12, "Properties."

ReturnKey

Syntax `returnKey`

Description The `returnKey` command sends a statement typed into the Message box to the current card. (If a field is open for text editing, pressing the Return key enters a return character.)

Notes The `returnKey` message, which invokes the `returnKey` command if it reaches HyperCard, is normally generated by pressing the Return key on the keyboard, but you can also send it from the Message box or execute it as a line in a script.

See also the `returnKey` system message in Table 8-3.

Save

Syntax `save stack` *stackName* `as` *fileName*

StackName is an expression that yields a valid stack name. *FileName* is an expression that yields a valid Macintosh filename.

Example

```
save stack "Pottery" as "NewPottery"
```

Description The `save` command saves a copy of the specified stack with the given filename. The stack is saved without a dialog box. Both *stackName* and *fileName* must be enclosed in quotation marks.

Note If the `save` command produces a file error, the error is stored in the HyperCard function `the result`. Other errors display error dialog boxes. For example, you would get an error dialog box if the filename you assign is the same as the name of a file that already exists.

Select

Syntax

```
select object
select [preposition] chunk of field
select [preposition] text of field
select empty
```

Object is an expression that yields the descriptor of a button or field, or `me`, or `target`; *preposition* is `before` or `after`; *chunk* is an expression that yields a chunk in a specified field, and *field* is an expression that yields the descriptor of a field. (Button and field descriptors and the special descriptor `me` are explained in Chapter 5. The special descriptor `target` is explained in Chapter 4.)

Examples

```
select button 1 -- chooses button tool and selects card
                -- button 1
select before char 1 of field 2 -- places insertion point at
                -- start of field
select after text of field 2 -- places insertion point at end
                -- of field
select char 1 to 5 of card field 2 -- selects first five
                -- characters of field
```

Description

The `select` *object* form chooses the appropriate tool and selects the object specified as though you had chosen the tool and clicked the object manually with the mouse. The other forms select text in the specified field. `Before` and `after` can be used to place the insertion point relative to the specified text or chunk of text. Using a chunk expression without a preposition selects the entire chunk, highlighting the characters in the chunk. The `select empty` form deselects highlighted text or removes the insertion point.

Set

Syntax

`set [the] property [of element] to value`

Property is a characteristic of a HyperCard object, menu, menu item, or window. *Element* is an expression that yields the descriptor of an object, menu, menu item, or window. *Value* is an expression that yields a valid setting for the particular property.

Examples

```
set name of field 1 to "Soccer"
set location of button "newButton" to the mouseLoc
set the visible of field 1 to "false" -- hide the field
set userLevel to 5 -- scripting
set the cmdChar of menuItem "Home Cards" of menu "Home" to 5
```

Description

The `set` command changes the state of a specified property. If the element to which the property belongs is not specified, the property must be a global or painting property.

Some properties cannot be changed with the `set` command. These exceptions are pointed out in the property descriptions in Chapter 12.

Script

The following example handler automatically draws a circle on the current card:

```
on mouseUp
   choose oval tool
   set linesize to 2
   set centered to true
   set dragspeed to 75   -- this changes the speed of
                         -- expansion
   drag from 155,70 to 285,200
   choose browse tool
end mouseUp
```

Notes

The properties of objects depend on the type of object. Generally, they are the characteristics shown in the Info dialog boxes under the Objects menu. All of the HyperCard properties are described in detail in Chapter 12, "Properties." See also the next command, `show`.

Show

Syntax

```
show menuBar
show object [at point]
show groups
show card picture
show picture of card
show background picture
show picture of background
show titleBar
show window windowName [at point]
show window stackName
```

Object yields one of the following objects:

a valid button descriptor in the current stack
a valid field descriptor in the current stack
`message [box]` or `message [window]`
`pattern window` or `window "patterns"` (the Patterns palette)
`tool window` or `window "tools"` (the Tools palette)
`message watcher` or `window "message watcher"`
`variable watcher` or `window "variable watcher"`
`window "navigator"` (the navigator palette)
`card window`

Point is an expression yielding two integers separated by commas representing the horizontal and vertical pixel offsets, respectively, on the screen. *Card* yields the descriptor of a card in the current stack. *Background* yields the descriptor of a background in the current stack. *WindowName* is the name of a window created with the `picture` command or custom external window. *StackName* is the name of an open stack.

Examples

```
show msg at 50,300
show tool window
show field "Names" at 1,1
show groups
show Message Watcher
```

Description

The `show` command displays a specified window or object at a specified location on the screen. If positioning offsets aren't given, the window or object is displayed at its previous location.

The `picture` form of the `show` command displays a graphic bitmap on the card or background that has been hidden with the `hide` command.

The `show groups` form of the `show` command makes a gray two-pixel underline below group-style text. The `show groups` command affects all group-style text in all fields globally.

The `show titleBar` form of the `show` command shows the stack window title bar, if it was hidden. If it wasn't hidden, it has no effect.

Script

The following example handler displays the Tools palette, the Patterns palette, and the Message box at their default locations when HyperCard first starts running:

```
on startUp
  show tool window
  show pattern window
  show msg
end startUp
```

Notes

The `show` command sets the `visible` and, optionally, `location` properties of the window or object.

If the menu bar is hidden and the screen is locked, `show menuBar` has no effect. See the `lock screen` and `unlock screen` commands in this chapter and the `lockScreen` property in Chapter 12, "Properties."

On Macintosh Plus and Macintosh SE screens, visible horizontal offsets range from 0 to 511, and visible vertical offsets range from 0 to 341. Members of the modular Macintosh family have variable visible offsets depending on the monitor currently in use.

`Message` can be abbreviated `msg`. `Background` can be abbreviated `bkgnd`. `Button` can be abbreviated `btn`. `Field` can be abbreviated `fld`.

For buttons and fields, *point* specifies the distance from the upper-left corner of the card window to the center of the button or field.

`Card window` refers to the current card window; *point* specifies the distance from the upper-left corner of the screen to the upper-left corner of the card window, disregarding the title bar at the top of the window. For the other windows, *point* specifies the distance from the upper-left corner of the card window to the upper-left corner of the other window, disregarding the drag bar at the top of the window.

The default location for the Message Watcher window is the lower-left corner of the screen. The default location for the Variable Watcher is the lower-right corner of the screen.

The menu bar always shows at the top of the screen.

You can use the `show` and `hide` commands with the navigator window only after it has been invoked with the `palette` command.

Valid window names for the `show` command are any of the windows returned by the `windows` function.

When you change a card window's `location` property with the `show card window at` *location* form, the system message `moveWindow` is sent. The `moveWindow` message is also sent when you drag the window to a new location, or zoom it in or out with the zoom box, causing the `location` property to change.

See also the `hide`, `palette`, and `set` commands in this chapter, the `show` system message in Table 8-3, the `windows` function in Chapter 11, and the `location` and `visible` properties in Chapter 12.

Show cards

Syntax

`show` *number* `cards`

Number is an expression yielding an integer, `marked`, or the word `all`.

Examples

```
show all cards
show ten cards
show 26 cards
show marked cards
show howMany cards -- howMany is a variable containing a number
```

Description

The `show cards` command displays the specified cards in the current stack in turn, beginning with the next card or, for the `show marked cards` form, the first marked card.

Script

The following example handler caches the cards in RAM when you open the stack, so that going to cards in the stack subsequently will be faster:

```
on openStack
   set lockScreen to true
   show all cards
   set lockScreen to false
end openStack
```

Notes

The `show all cards` form shows all cards in the stack. HyperCard doesn't send the `openCard` system message when a card is displayed by `show cards`, nor do visual effects occur. After the cards are shown, the last one shown (where you began in the case of `show all cards`) is the current card.

See also the `marked` property in Chapter 12.

Sort

Syntax

```
sort [lines|items of] container [sortDirection] [sortStyle]
sort [[[marked] cards of [this]] stack] [sortDirection] ¬
[sortStyle] by expression
sort [[[marked] cards of] [background] [sortDirection] ¬
[sortStyle] by expression
```

Container is a field expression, variable, or the word `it`. *SortDirection* is `ascending` or `descending`, *sortStyle* is `text`, `numeric`, `dateTime`, or `international`, and *expression* is any expression. *Background* is an expression that yields a background descriptor.

Examples

```
sort numeric by second word of field 1
sort descending text by last word of field "Names"
sort cards of this stack by field "Names"
sort marked cards descending numeric by bkgnd field 2
sort marked cards of background "Notes" by bkgnd field 2
sort this background by field 1
sort lines of field 3 ascending
sort items of field 3 dateTime
sort field 3
sort It numeric
```

Description

The `sort` command orders the lines or items in a container. The `sort` command also orders all the cards or marked cards in a single background or a stack according to the value of *expression*, which is evaluated individually for each card in the stack.

The default value for *sortDirection* is `ascending`, and the default *sortStyle* is `text`. If you do not specify `lines` or `items of` when sorting a container, the default is `lines`.

Script

The following example handler shuffles the cards in a stack randomly when the user goes to it from another stack:

```
on openStack
   sort numeric by random(the number of cards)
end openStack
```

Notes

The `dateTime` style sorts the stack using one of the forms of date or time (shown with the `convert` command, in this chapter), with earliest placed first in the ascending direction. The `international` style assures correct sorting of non-English text containing diacritical marks and special characters, depending on the international resources in your System file, your version of HyperCard, the Home stack, and the current stack. The `dateTime` style also works correctly with non-English forms of date and time modified by international resources in the System file.

See also the `marked` property in Chapter 12, "Properties," and the `mark` command in this chapter.

Start using

Syntax

```
start using stack stackName
```

StackName is an expression that yields a stack name.

Example

```
start using stack "HD80:myStack"
```

Description

The `start using` command inserts the specified stack between the current stack and the Home stack in the message-passing hierarchy. Each successive stack that is added to the message-passing hierarchy is inserted just after the current stack. If a stack that is already in use is "used" again, its previous place in the message-passing hierarchy changes to the place just after the current stack.

The `start using` command allows you to use the stack script and resources of any other HyperCard stack, not just the Home stack. Once the `start using` statement is executed, the handlers in the script of the specified stack and the XCMDs and other resources in its resource fork are available for use.

Script

To change the message-passing hierarchy when opening a stack, simply place a `start using` statement in the `openStack` handler.

```
on openStack
  start using stack "myStack:ScriptStack"
end openStack
```

To remove the stack from the hierarchy when closing a stack, place a `stop using` statement in the `closeStack` handler.

```
on closeStack
  stop using stack "myStack:ScriptStack"
end closeStack
```

Note

See also the description of the message-passing hierarchy in Chapter 4 and the next command, `stop using`.

Stop using

Syntax `stop using stack` *stackName*

StackName is an expression that yields a stack name.

Example
```
stop using stack "HD80:myStack"
```

Description The `stop using` command removes the specified stack from the message-passing hierarchy.

Script To remove the stack from the message-passing hierarchy when closing a stack, place a `stop using` statement in the `closeStack` handler.

```
on closeStack
   stop using stack "myStack:ScriptStack"
end closeStack
```

Note See also the description of the message-passing hierarchy in Chapter 4 and the previous command, `start using`.

Subtract

Syntax `subtract` *number* `from [`*chunk* `of]` *container*

Number is an expression that yields a number. *Chunk* is an expression that yields a chunk of a container. *Container* is an expression that yields a field, the Message box, the selection, or a variable.

Examples
```
subtract 2 from It
subtract field 1 from field 2
```

Description The `subtract` command subtracts the value of *number* from the value of *container*, leaving the result in the container. The value previously in the container must be a number; it is replaced with the new value.

TabKey

Syntax

```
tabKey
```

Description

The `tabKey` command opens the first unlocked field on the current background or card (placing the text insertion point in the field) and selects its entire contents. If a field is already open, `tabKey` closes it and opens the next field, selecting its contents.

Script

The following example handler sets the insertion point in the first field when the card is opened, so that the user can type something:

```
on openCard
  tabKey
end openCard
```

Notes

The `tabKey` message, which invokes the `tabKey` command if it reaches HyperCard, is normally generated by pressing the Tab key on the keyboard. But you can also send it from the Message box or execute it as a line in
a script.

The `tabKey` command opens fields in the following order: from the lowest number to the highest, through the background fields first, then through the card fields.

See also the `tabKey` system message in Table 8-3.

Type

Syntax

`type` *expression* `[with` *key*`[,` *key2* `[,` *key3*`]]]`

Text is an expression that yields a text string, and *key*, *key2*, and *key3* are one of the following key names, separated by commas: `shiftKey`, `optionKey`, or `commandKey` (or `cmdKey`).

Examples

```
type "Now is the time for all good persons."
type "p" with commandKey -- print card
```

Description

The `type` command enters the value of *text* at the text insertion point, as though you had typed it manually. If the `with commandKey` form is used, no text appears at the insertion point, rather the action defined for the Command-key combination is carried out.

Script

The following example handler chooses the Browse tool, clicks at the center of the specified field, and types a literal string:

```
on autoType
   choose browse tool
   click at the loc of field "whereToType"
   type "Automatic writing apppears before your eyes..."
end autoType
```

Notes

The text insertion point is placed by clicking in an unlocked field with the Browse tool or by sending the `tabKey` message. Manipulating the text insertion point is described in the *HyperCard Reference*.

Paint text can be typed at the text insertion point on a card or background with the Paint Text tool selected.

Unlock screen

Syntax

`unlock screen [with` *effectName*`]`

EffectName is an expression that yields any of the effect names described under the `visual` command later in this chapter.

Example

```
unlock screen with fade to 4
```

Description

The `unlock screen` command sets the `lockScreen` property to `false`, allowing HyperCard to update the screen. In addition, the `with` *visualEffect* option specifies a single visual transition that occurs as the screen is updated.

Visual effects can't be compounded using `unlock screen`, as they can be using the `visual` command. Visual effects compounded by the `visual` command are not executed until a `go` command is encountered. HyperCard flushes unexecuted visual effects and sets `lockScreen` to `false` at idle time (in effect, at the end of all pending handlers).

Note

See also the `visual` command in this chapter and the `lockScreen` property in Chapter 12.

Unmark

Syntax

```
unmark card
unmark cards where expression
unmark all cards
unmark cards by finding text [in field]
unmark cards by finding chars text [in field]
unmark cards by finding string text [in field]
unmark cards by finding whole text [in field]
unmark cards by finding word text [in field]
```

Card is an expression that yields a card descriptor. *Expression* is an expression that yields the criteria on which you want to base the unmarking of cards. *Text* is an expression that yields the text criteria on which you want to base the marking of cards. *Field* is an expression that yields a field descriptor.

Examples

```
unmark the next card
unmark cards where "We be shaking" is in field 2
unmark all cards
unmark card by finding word fire in bkgnd field 3
```

Description

The `unmark` command sets the `marked` property for the specified card or cards to `false`. The `marked` property of a card can also be changed with the Card Marked option in the Card Info dialog box. By default, the `marked` property of a card is `false`.

The `by finding` form of the `mark` command uses `chars`, `word`, `whole`, and `string` to define the search criteria the same way the `find` command does. See the description of the `find` command for information about how to use these forms.

The `unmark` command can be used with the `mark` command in searches where you want to find and mark cards containing particular information while excluding other unnecessary information. See the description and script example used for the `mark` command, which is described earlier in this chapter.

Note

See also the `marked` property in Chapter 12, "Properties," and the `mark` command, earlier in this chapter.

Visual

Syntax

```
visual [effect] effectName [speed] [to image]
```

EffectName is one of the following:

```
barn door close      shrink to top | center | bottom
barn door open       stretch from top | center | bottom
checkerboard         venetian blinds
dissolve             wipe down
iris close           wipe left
iris open            wipe right
plain                wipe up
scroll down          zoom close
scroll left          zoom in
scroll right         zoom open
scroll up            zoom out
```

Speed is one of the following:

```
fast                 very fast
slow[ly]             very slow[ly]
```

Image is one of the following:

```
black                inverse
white                card
gray
```

Examples

```
visual effect barn door open
visual dissolve slowly to white
```

Description

The `visual` command specifies a visual transition for HyperCard to use the next time it opens a card, as the current card is closed. The default `plain` visual effect causes all of the current image to be replaced immediately by the image of the next card. If you use the `to` *image* form, the visual effect occurs as a transition from the current card to a completely white, gray, or black screen image, to the inverted image of the current card, or to the image of the next card; `to card` is the default.

Script

The following example handler stacks two visual effects, which occur in succession, so that the transition appears as a fade to black, then to the next card:

```
on fadeOut
   visual effect dissolve to black
   visual effect dissolve to card
   go next card
end fadeOut
```

Notes

Visual effects don't happen when you use the arrow keys or the `show cards` command to change cards; they occur only when the `go` is executed, so they must be set up in a handler that also contains a `go` command. If a `go` command is not executed, visual effects set up in the handler are canceled when the handler finishes executing.

You can stack up several visual effects that will occur one after the other when you go to the next card.

See also the `unlock screen` command, earlier in this chapter.

Wait

Syntax

```
wait [for] time [seconds]
wait until condition
wait while condition
```

Time is an expression that yields an integer and *condition* is an expression that yields `true` or `false`.

Examples

```
wait 60 seconds
wait until the mouse is down
```

Description

The `wait` command causes HyperCard to pause before executing the rest of the handler, either for a specific length of time, until a specified condition becomes true, or while a specified condition remains true.

Script

The following example handler allows time to view each card:

```
on slideshow
  repeat the number of cards
    visual effect dissolve slowly
    go next card
    wait 2 seconds
  end repeat
end slideshow
```

Note

If `seconds` is not specified for *time*, HyperCard uses `ticks` (1/60 second), which can also be specified explicitly.

Write

Syntax

`write` *text* `to file` *fileName* [*start*]

Text is an expression that yields text, and *fileName* is an expression that yields a filename. *Start* is a positive integer that represents the starting point (in characters from the beginning of the file) at which you want to write the text.

Examples

```
write field "address" to file "myDisk:myFile"
write "first line" & return & "second line" to file "twoliner"
```

Description

The `write` command causes HyperCard to copy the specified text into the specified disk file. Optionally you can specify the starting point at which to write the text.

Script

The following example handler opens a file specified in a global variable, writes the entire contents of the specified field to the file starting at character 5, then closes the file:

```
on writeFile
  global filename
  open file filename
  write background field 1 to file filename
  close file filename
end writeFile
```

Notes

The file must have been opened previously with the `open file` command and should be closed, when copying is completed, with the `close file` command.

The first `write` command that you execute after opening the file replaces any previous contents. Subsequent `write` commands append text to what's already been written.

You must provide the full pathname of the file if it's not at the same directory level as HyperCard. (See "Identifying a Stack" in Chapter 5 for an explanation of pathnames.)

If the file is locked or its disk is full, HyperCard displays an error dialog box and closes the file. HyperCard automatically closes all open files when an `exit to HyperCard` statement is executed, when you press Command-period, or when you quit HyperCard.

See also the `close file`, `open file`, and `read` commands, in this chapter.

Chapter 11 **Functions**

This chapter describes HyperTalk built-in functions.

A **function** is a named value that is calculated by HyperCard when the statement it is in executes. The value of a function changes according to conditions of the system or according to values of parameters that you pass to the function when you use it. When HyperCard reads a function name in a line of HyperTalk, it places the function's current value—its result—in that location before completing other actions.

Function calls

To make a function call, that is, to use it in a HyperTalk statement, you must either use the word `the` before the function name or append parentheses after it. If a single parameter is passed to a function, the parameter can be enclosed in the parentheses or can follow the word `of`. (When `of` is used in this way to indicate the function call, the word `the` preceding the function name is optional.) If more than one parameter is passed to a function, all parameters must be enclosed in the parentheses and separated from each other by commas. Here are some examples of function calls:

```
put the time into msg
put time() into background field "Time"
put the length of myVariable into card field "howLong"
put average(total_1,total_2,total_3) into Projection
get the clickChunk
```

You can define your own functions in HyperTalk using the function handler structure described in Chapter 9.

- *User-defined functions override built-in ones with the same name:* If you define your own function having the same name as a built-in one, yours overrides the built-in one if the function call is made with the parentheses syntax (unless the function call is made farther along the hierarchy than the handler's script).

You can call the built-in functions of HyperCard directly and bypass any user-defined functions by using the word `the` before the function name. You can also use `of`, rather than using the parentheses syntax; however, functions having more than one parameter always require parentheses.

Syntax description notation

The terms *factor* and *expression* are defined in Chapter 7. Briefly, a factor can be a constant, literal, function, property, number, or container, and an expression can be a single factor or a complex expression built with factors and operators. Also, a factor can be an expression within parentheses. The term *yields* indicates a specific kind of value, such as a number or a text string, that must result from evaluation of an expression when a restriction applies. (For example, the factor or expression used with the `abs` function must yield a number.) However, any HyperTalk value can be treated as a text string.

To help you quickly locate functions that are new or enhanced since the first edition of this book, the names of the new and enhanced functions are followed with a graphic like the one at the beginning of this paragraph.

Abs

Syntax

```
the abs of factor
abs(expression)
```

Factor and *expression* yield numbers.

Example

```
put abs(a-b) into field "theOffset"
```

Description

The `abs` function returns the absolute value (makes the sign positive) of the number passed to it.

Annuity

Syntax

```
annuity(rate, periods)
```

Rate and *periods* are expressions that yield numbers

Examples

```
put myPayment*annuity(.015,12) into presentValue
put myPayment*annuity(.015,12)*compound(.015,12) into futureValue
```

Description

The `annuity` function is used to compute the present or future value of an ordinary annuity. *Rate* is the interest rate per period, and *periods* is the number of periods over which the value is calculated. The formula for `annuity` is

`annuity(`*rate, periods*`) = (1-(1+`*rate*`)` ^{periods}`) / `*rate*

The `annuity` function is more accurate than computing the formula above using basic arithmetic operations and exponentiation, especially when *rate* is small.

Note

See also the `compound` function, later in this chapter.

Atan

Syntax

```
the atan of factor
atan(expression)
```

Factor and *expression* yield numbers.

Example

```
put atan(1.0) into field "arcTan" -- yields 0.785398
```

Description

The `atan` function returns the trigonometric arc tangent (inverse tangent) of the number passed to it; that is, the angle whose tangent is equal to the given value. The result is expressed in radians.

Radians can be converted to degrees by multiplying by 180 and dividing the result by the value of the constant `pi`.

Script

The following example handler converts a value in radians to degrees and puts the result into the Message box:

```
on radiansToDegrees var
  put round((atan(var)*180)/pi) into msg
end radiansToDegrees
```

Average

Syntax

`average (`*list*`)`

List is a sequence of comma-separated expressions that yield numbers, or it is a single container that contains such a sequence.

Example

```
put average(1,2,3) into field "avg"
```

Description

The `average` function returns the average of the numbers passed to it.

Script

The following example handler displays the average of a list of numbers contained in one line of a field:

```
on avgSupplyPrice
  put "12.95,10.50,14.75,15.00,9.95" into line 3 of field¬
  "suppliers"
  answer "Average widget cost:" && average (line 3 of field¬
  "suppliers")
end avgSupplyPrice
```

CharToNum

Syntax

`the charToNum of` *factor*

`charToNum (`*expression*`)`

Factor and *expression* yield a character.

Example

```
put the charToNum of "a" into It -- yields 97
```

Description

The `charToNum` function returns an unsigned integer representing the ASCII equivalent value of the character passed to it.

Notes

If more than one character is passed, `charToNum` returns the ASCII value of the first character. If *factor* is a literal, it must appear within quotation marks. See also the `numToChar` function, later in this chapter.

ClickChunk

Syntax

```
the clickChunk
clickChunk()
```

Examples

```
put the clickChunk into card field "ExpressMyClick"
get the clickChunk
```

Description

The `clickChunk` function returns a chunk expression referring to the text clicked in a field, and is typically something like `char 1 to 3 of bkgnd field 3`.

`ClickChunk` refers to the single word clicked, with the definition of a word being any characters delimited by white space (commas, tabs, spaces, returns, and so on.). If the location clicked happens to have the style `group`, then the largest contiguous run of text that has the group style is returned, making it possible to refer to ranges or phrases rather than just single words. `Group` is a possible value of the `textStyle` property.

Script

Because chunks of text also have properties, you could use the `clickChunk` function to examine the `textstyle` property of a chunk of text in a field, and then take the appropriate action based on that style of text. For example, you might have an application that provides definitions for any bold words when one of the words is clicked.

The following handler placed in a script of a locked field examines the style of a chunk of text clicked in that field and, if that text is bold, calls up another field containing definitions.

```
on mouseUp
  if the textStyle of the clickChunk is bold
  then show card field "definitions"
end mouseUp
```

Note

See also the `clickLine`, `clickLoc`, and `clickText` functions in this chapter and the `textStyle` property in Chapter 12, "Properties."

ClickLine

Syntax

```
the clickLine
clickLine()
```

Examples

```
put the clickLine into card field "ExpressMyClick"
get the clickChunk
```

Description

The `clickLine` function returns the line (based on actual return characters, not display lines) that was clicked. A typical result is `line 5 of card field 2`. One sentence or line of text may not fit in the width of a field and have to wrap onto subsequent lines on the display. It may appear as though more than one line is in the field, but if the lines are not delimited with return characters, `clickLine` returns an expression like `line 1 of card field 5` when a user clicks within the text in that field. If you want `clickLine` to return a unique line number for each line that is displayed within a field, be sure to end each line in the field with a return character.

Note

See also the `clickChunk`, `clickLoc`, and `clickText` functions in this chapter.

ClickLoc

Syntax

```
the clickLoc
clickLoc()
```

Example

```
put the clickLoc into card field "firstClick"
```

Description

The `clickLoc` function returns the point on the screen where the user most recently clicked before the handler started executing. The location is determined at the time the message is first sent—the mouse could be elsewhere by the time the message is received. The location point is returned as two integers separated by a comma, representing horizontal and vertical pixel offsets from the upper-left corner of the card.

Script

The following example handler, when it is in the script of a locked field, selects a word in the field when the user clicks the word:

```
on mouseUp
   set locktext of me to false -- unlock the locked field
   -- next two lines double-click on the location
   click at the clickLoc
   click at the clickLoc
   put "You clicked on the word:" && the selection
   set lockText of me to true -- must lock it again when we leave
end mouseUp
```

ClickText

Syntax

```
the clickText
clickText()
```

Examples

```
put the clickText into card field "ExpressMyClick"
get the clickText
```

Description

`ClickText` returns the text of a single word clicked, with the definition of a word being any characters delimited by white space (commas, tabs, spaces, returns, and so on). If the location clicked happens to have the style `group`, then the largest contiguous run of text that has the group style is returned, thus making it possible to refer to and analyze ranges or phrases rather than just single words. `Group` is a possible value of the `textStyle` property.

Script

The `clickText` function can be used to implement glossary lookup or provide other hypertext-type functions with something like the following handler, which should be placed in the script of the field being clicked:

```
on mouseUp
  get the clickText -- must be done before leaving card
  lock screen
  go stack "Glossary"
  find it in field "Words" -- is it a glossary entry?
  if the result = "Not found" then go back
  else unlock screen with dissolve -- display glossary entry
end mouseUp
```

Note

See also the `clickChunk`, `clickLoc`, and `clickLine` functions in this chapter and the `textStyle` property in Chapter 12, "Properties."

CommandKey

Syntax

```
the commandKey
commandKey()
```

Example

```
if the commandKey is up then put "Wow" into msg box
```

Description

The `commandKey` function returns the constant `up` if the Command key is not pressed or `down` if it is pressed.

Notes

The `commandKey` function name can be abbreviated `cmdKey`.

See also the `optionKey` and `shiftKey` functions, later in this chapter.

Compound

Syntax

`compound(`*rate*`,` *periods*`)`

Rate and *periods* are expressions that yield numbers.

Examples

```
put futureValue/compound(.10,12) into presentValue
put presentValue*compound(.10,12) into futureValue
```

Description

The `compound` function is used to compute the present or future value of a compound interest–bearing account. *Rate* represents the interest rate per period, and *periods* is the number of periods over which the value is calculated. The formula for `compound` is

`compound(`*rate*, *periods*`) = (1+`*rate*`)` ^{periods}

The `compound` function is more accurate than computing the formula expression above using standard arithmetic operations and exponentiation, especially when *rate* is small.

Script

The following example handler calculates the value in one year of an account earning 7½ percent interest compounded monthly:

```
on calcInterest
   ask "Enter the beginning balance:" with empty
   set numberFormat to ".00" -- dollars and cents format
   put "Value in 1 year $" & it * compound(.075/12,12)
end calcInterest
```

Note

See also the `annuity` function, earlier in this chapter.

Cos

Syntax

```
the cos of factor
cos (expression)
```

Factor and *expression* yield numbers.

Example

```
put the cos of 2 -- puts -.416147 into the Message box
```

Description

The `cos` function returns the cosine of the angle which is passed to it. The angle must be expressed in radians.

Note

Radians can be converted to degrees by multiplying by 180 and dividing the result by the value of the constant `pi`.

Date

Syntax

```
the [adjective] date
```

Adjective is `long`, `short`, or `abbreviated` (or `abbrev` or `abbr`).

Example

```
put last word of the long date into background field "Year"
```

Description

The `date` function returns a string representing the current date set in your Macintosh. There are three forms of the `date` function. Here are examples of the format used by each:

```
the short date      10/17/89
the long date       Tuesday, October 7, 1989
the abbrev date     Tue, Oct 17, 1989
```

Without an adjective, the `date` function returns the short date.

Script

The following example handler puts the current date into a field when another field (whose script contains the handler) is changed:

```
on closeField
  put the long date into field "lastUpdate"
end closeField
```

DiskSpace

Syntax

```
the diskSpace
diskSpace()
```

Example

```
if the diskSpace < 100000 then answer ¬
"Your disk is getting full."
```

Description

The `diskSpace` function returns an integer representing the number of bytes of free space on the disk that contains the current stack.

Script

The following function handler is used by the second handler (for the `writeFile` message) to ensure that there is enough space on a disk to write to a file on that disk:

```
function checkSpace var
  if the diskSpace > var then return "OK" ¬
  else return "FULL"
end checkSpace

on writeFile
  global var
  put "MyFilename" into filename
  if checkSpace(card field 1) is "OK" then
    open file filename
    write var to file filename
    close file filename
  else answer "Can't write that file; the disk is full."
end writeFile
```

Exp

Syntax

```
the exp of factor
exp(expression)
```

Factor and *expression* yield numbers.

Example

```
put the exp of 2 -- puts 7.389056 into the Message box
```

Description

The `exp` function returns the mathematical exponential of its argument (the constant *e*, which equals 2.7182818, raised to the power specified by the argument).

Exp1

Syntax

```
the exp1 of factor
exp1(expression)
```

Factor and *expression* yield numbers.

Example

```
put the exp1 of 2 -- puts 6.389056 into the Message box
```

Description

The `exp1` function returns 1 less than the mathematical exponential of its argument (1 less than the result of the constant *e* raised to the power specified by the argument). That is, it computes

```
exp(number) - 1
```

Exp2

Syntax

```
the exp2 of factor
exp2(expression)
```

Factor and *expression* yield numbers.

Example

```
put the exp2 of 16 -- puts 65536 into the Message box
```

Description

The `exp2` function returns the value of 2 raised to the power specified by the argument.

FoundChunk

Syntax

```
the foundChunk
foundChunk()
```

Example

```
put the foundChunk
```

Description

The `foundChunk` function returns a chunk expression describing the location of the text found in a field with the `find` command. For example, if field 1 contained *Now is the time,* the commands (placed inside a handler)

```
find "Now"
put the foundChunk
```

would put `char 1 to 3 of bkgnd field 1` into the Message box.

Note

See also the `find` command in Chapter 10.

FoundField

Syntax

```
the foundField
foundField()
```

Example

```
put the foundField
```

Description

The `foundField` function returns the descriptor of the field in which the text was found with the `find` command. The result is in a form such as `card field 1`.

Script

The following handler uses `foundField` to put the field descriptor of a field containing the specified word in the Message box.

```
on getField theWord
   lock screen
   push card
   find theWord
   if the result is "not found" then
     put "It's not here"
   else
     put the number of this card into cardNum
     put "Your word was found in" && the foundField && ¬
     "of card number" && cardNum
   pop card
   unlock screen
end getField
```

To make the script work, put it in the stack or background script, then enter `getField` followed by the word you want to find in the Message box.

FoundLine

Syntax

```
the foundLine
foundLine()
```

Example

```
put the foundLine
```

Description

The `foundLine` function returns a chunk expression describing the line in which the beginning of the text was found with the `find` command. The result is in a form such as `line 1 of card field 2`.

FoundText

Syntax

```
the foundText
foundText()
```

Example

```
put the foundText
```

Description

The `foundText` function returns the characters that are enclosed in the box after the `find` command has executed successfully; for example, the commands

```
find "Hyper"
put the foundText
```

would put `HyperCard` in the Message box if it were the word containing the matching string.

Length

Syntax

```
the length of factor
length(expression)
```

Factor and *expression* yield text strings.

Examples

```
put length("tail") into It -- yields 4
if the length of word n of field 5 > 25
then add 1 to fogIndex
```

Description

The `length` function returns the number of characters (including spaces, tabs, and return characters) in the text string passed to it.

Note

If *expression* is a literal, it must appear within quotation marks. The `length` function is identical in effect to the following form of the `number` function:

```
the number of characters in factor
```

Ln

Syntax

```
the ln of factor
ln(expression)
```

Factor and *expression* yield numbers.

Example

```
put the ln of 10 -- puts 2.302585 into Message box
```

Description

The `ln` function returns the base-*e* (natural) logarithm of the number passed to it.

Ln1

Syntax

```
the ln1 of factor
ln1(expression)
```

Factor and *expression* yield numbers.

Example

```
put the ln1 of 10 -- puts 2.397895 into Message box
```

Description

The `ln1` function returns the base-*e* (natural) logarithm of the sum of 1 plus the number passed to it. That is, it computes

```
ln(1+number)
```

If *number* is small, the result is more accurate than using standard arithmetic operations.

Log2

Syntax

```
the log2 of factor
log2(expression)
```

Factor and *expression* yield numbers.

Example

```
put the log2 of 10 -- puts 3.321928 into the Message box
```

Description

The `log2` function returns the base-2 logarithm of the number passed to it.

Max

Syntax

`max(`*list*`)`

List is a sequence of comma-separated expressions that yield numbers, or it is a single container that contains such a sequence.

Example

```
put max(5,10,7.3) -- puts 10 into the Message box
```

Description

The `max` function returns the highest-value number from a list of numbers passed to it. If the source of the list is a container with more than one line in it, only the first line is used.

Script

The following example handler displays the highest number in a list contained in a variable:

```
on highStock
  put "12.50,10,7.95,14.76,13.70" into stockPrices
  answer "The highest price for the month is:" ¬
  && max(stockPrices)
end highStock
```

Menu

Syntax

```
menu menu
```

Menu is an expression that yields the name or number (integer or ordinal number) of a standard HyperCard menu or a custom menu in the menu bar.

Examples

```
put menu 3
put menu Home into card field 3
```

Description

The `menu` function returns a return-delimited list of all the menu items in the specified menu, including the Apple menu and any custom menus.

Notes

See also the `create menu` command in Chapter 10 and the `menus` and `number` functions in this chapter.

If you precede the function `menu` with the word `the`, HyperCard returns an error.

Menus

Syntax

```
the menus
menus()
```

Examples

```
put the menus into card field 2
put menus() into MyVar -- MyVar is a variable
```

Description

The `menus` function returns a return-delimited list of all the menus currently in the HyperCard menu bar, including the Apple menu and any custom menus.

Note

See also the `create menu` command in Chapter 10.

Min

Syntax

`min(`*list*`)`

List is a sequence of comma-separated expressions that yield numbers, or it is a single container that contains such a sequence.

Example

```
put min(5,10,7.3) -- puts 5 into the Message box
```

Description

The `min` function returns the lowest-value number from a list of numbers passed to it. If the source of the list is a container with more than one line in it, only the first line is used.

Script

The following example handler displays the lowest number in a list contained in a variable:

```
on lowStock
  put "12.50,10,7.95,14.76,13.70" into stockPrices
  put "The lowest price for the month is:" ¬
  && min(stockPrices)
end lowStock
```

Mouse

Syntax

```
the mouse
mouse()
```

Example

```
if the mouse is up put "Press the mouse button" into msg
```

Description

The `mouse` function returns the constant `up` if the mouse button is not pressed, `down` if it is pressed.

Script

The following example handler determines whether the user has single-clicked or double-clicked the button whose script contains the handler:

```
on mouseUp
  put the ticks into start
  repeat until the ticks-start > 4 -- adjust for click speed
    if the mouse is "down" then
      go last card -- put your double-click action here
      exit mouseUp
    end if
  end repeat
  go next card -- put your single-click action here
end mouseUp
```

MouseClick

Syntax

```
the mouseClick
mouseClick()
```

Example

```
if the mouseClick then put the mouseLoc
```

Description

The `mouseClick` function determines if the mouse button is clicked. If no click is sensed, `the mouseClick` immediately returns the constant `false`. `MouseClick` returns the constant `true` when the mouse button is clicked. `MouseClick` does not return `true` more than one time for a given mouse click.

Script

The following example handler demonstrates operation of the `mouseClick` function by informing the user whether or not it sensed a click during its execution:

```
on mouseUp
  put "Click or don't click..."
  wait 5 seconds
  if the mouseClick then
    put "You clicked."
  else
    put "You didn't click."
  end if
end mouseUp
```

MouseH

Syntax

```
the mouseH
mouseH()
```

Example

```
if the mouseH > 319 put "Stop" into msg
```

Description

The `mouseH` function returns an integer representing the number of horizontal pixels from the left side of the card to the current location of the mouse pointer.

MouseLoc

Syntax

```
the mouseLoc
mouseLoc()
```

Example

```
show button "alwaysReady" at the mouseLoc
```

Description

The `mouseLoc` function returns the point on the screen where the pointer is currently located. This point is returned as two integers separated by a comma, representing horizontal and vertical pixel offsets from the upper-left corner of the card.

Script

The following example handler, in a button script, allows the user to drag the button around the screen:

```
on mouseDown
  repeat until the mouse is up
    show the name of me at the mouseloc
  end repeat
end mouseDown
```

MouseV

Syntax

```
the mouseV
mouseV()
```

Example

```
if the mouseV > 199 put "Stop" into msg
```

Description

The `mouseV` function returns an integer representing the number of vertical pixels from the top of the card to the current location of the pointer.

Number

Syntax

```
[the] number of objects
[the] number of chunks in expression
[the] number of backgrounds [in this stack]
[the] number of cards in background
[the] number of cards [in this stack]
[the] number of marked cards
[the] number of menus
[the] number of menuItems of menu
[the] number of windows
```

Objects is `[background] buttons`, `[card] fields`, `backgrounds`, or `cards`. *Chunks* is `characters` (or `chars`), `words`, `items`, or `lines`, and *expression* yields a container or text string. *Background* is an expression that yields the descriptor of a background in the current stack. *Menu* is an expression that yields a menu descriptor.

Examples

```
put the number of buttons into It
put number of items of line 1 of field 2 into listSize
put the number of chars in msg into line 3 of field 2
if number of chars in myVar > 10 then put "Big" into msg
get the number of cards of bkgnd 3
```

Description

The `number` function returns the number of buttons or fields on the current card or on its background, the number of backgrounds or cards in the current stack, the number of chunks of a specified kind in a designated container or text string, the number of cards that are associated with a specified background, the number of marked cards in the current stack, the number of menus in the menu bar, the number of menu items in a specified menu, or the number of windows currently available to HyperCard.

Script

The following example handler uses the `number` function to delete all the card fields on a card, regardless of how many there are:

```
on deleteFields
   put the tool into oldTool
   choose field tool
   repeat with whichField = the number of card fields down to 1
      -- you must count down like this, not up
      click at the loc of card field whichField
      doMenu clear field
   end repeat
   choose oldTool
end deleteFields
```

Notes

If `backgrounds` is not specified with `buttons`, the number of card buttons is returned; if `card` is not specified with `fields`, the number of background fields is returned. If the `number` function is used with a chunk name, it returns the number of chunks of that kind within the designated container or other factor yielding a text string.

The factor can be a chunk expression, so you can get the number of chunks of one kind within another chunk:

```
the number of chars in first word of field 1
```

You can also use the syntax that uses parentheses with the `number` function. For example,

```
number(cards)
number(menus)
number(bkgnds)
number(fields)
```

Backgrounds can be specified with the abbreviation `bkgnds`.

See also the `number` and `marked` properties in Chapter 12.

NumToChar

Syntax

```
the numToChar of factor
numToChar(expression)
```

Factor and *expression* yield positive integers.

Example

```
put numToChar(67) into word 4 of line 9 of field "ASCII Chart"¬
-- yields C
```

Description

The `numToChar` function returns the character whose ASCII equivalent value is that of the integer passed to it.

Script

The following example handler turns all of the lowercase letters in a field into uppercase letters:

```
on uppercase
  put card field 4 into temp
  -- variables are faster than fields
  repeat with count = 1 to the length of temp
    get character count of temp
    if charToNum of It > 96 and charToNum of It < 123 then
      put numToChar(charToNum(It)-32) into character count¬
      of temp
    end if
  end repeat
  put temp into card field 4
end uppercase
```

Notes

See also the `charToNum` function, earlier in this chapter.

Offset

Syntax

`offset (`*string1, string2*`)`

String1 and *string2* are both expressions yielding text strings.

Examples

```
put offset("hay",field 1) into the Message box
offset("a","abc") -- typed in msg, returns 1
```

Description

The `offset` function returns the number of characters from the beginning of the *string2* string at which *string1* begins. If *string1* doesn't appear within *string2*, 0 is returned.

Script

The following function handler finds every occurrence of a string within a container, and it replaces every occurrence with a second string:

```
function searchAndReplace container,original,replacement
  repeat until original is not in container
  -- loop until all are replaced
    put offset(original,container) into start
    -- set start to location of original
    put replacement into char start to start + ¬
    (the length of original - 1) of container
  end repeat
  return container
end searchAndReplace
```

Note

The parameters passed to the `offset` function can both be arithmatic or logical (as well as text) expressions; after evaluation, the results are treated as strings.

OptionKey

Syntax

```
the optionKey
optionKey()
```

Example

```
if the optionKey is down then choose button tool
```

Description

The `optionKey` function returns the constant `up` if the Option key is not pressed, `down` if it is pressed.

Note

See also the `commandKey` and `shiftKey` functions in this chapter.

Param

Syntax

`the param of` *factor*

`param(`*expression*`)`

Factor and *expression* yield integers.

Example

```
if param(1) is empty then answer "Message has no parameters."
```

Description

The `param` function returns a parameter value from the parameter list passed to the currently executing handler. The parameter returned is the *n*th parameter where *n* is the integer derived from *factor* or *expression.* The value of `param(0)` is the message name.

Script

The following example handler sums the numeric arguments passed to it, regardless of how many there are:

```
on addUp -- adds a variable number of arguments
   put 0 into total
   repeat with i = 1 to the paramCount
      add param(i) to total
   end repeat
   put total
end addUp
```

Note

See also the `paramCount` and `params` functions, in this chapter, and the discussion of parameter passing in Chapter 4, "Handling Messages."

ParamCount

Syntax

```
the paramCount
paramCount()
```

Example

```
if the paramCount < 3 then
     put "I need at least three arguments."
```

Description

The `paramCount` function returns the number of parameters passed to the currently executing handler.

Script

The following example handler draws an oval differently depending on the number of parameters passed to it:

```
on drawOval
  if the paramCount is 1 then
  -- if 1 param use it as oval size and use default line size
    choose oval tool
    drag from 30,30 to 30 + param(1),30 + param(1)
  else if the paramCount is 2 then
  -- if 2 params use the second as line size
    choose oval tool
    set lineSize to param(2)
    drag from 30,30 to 30 + param(1),30 + param(1)
  end if
  choose browse tool
  reset paint
end drawOval
```

Note

See also the `param` and `params` functions, in this chapter, and the discussion of parameter passing in Chapter 4, "Handling Messages."

Params

Syntax

```
the params
params()
```

Example

```
put the params into field "messageReceived"
```

Description

The `params` function returns the entire parameter list, including the message name, passed to the currently executing handler.

Script

The following example handler is useful primarily for debugging, to see if the parameters passed to a handler are correct:

```
on myMessage
  put the params
  -- rest of myMessage handler goes here
end myMessage
```

Note

See also the `param` and `paramCount` functions, in this chapter, and the discussion of parameter passing in Chapter 4, "Handling Messages."

Random

Syntax

```
the random of factor
random(expression)
```

Factor and *expression* yield positive integers.

Example

```
set the loc of button "jumpy" to random(320),random(200)
```

Description

The `random` function returns a random integer between 1 and the integer derived from *factor* or *expression*, inclusive. `Random` supports integers up to $2^{31}-2$.

Script

The following example handler draws 10 unique random numbers between 1 and 100:

```
on mouseUp
  global randomList
  put empty into randomList
  repeat until the number of items in randomList is 10
    get random(100)
    if It is not in randomList
    then put It & "," after randomList
  end repeat
  delete last char of randomList -- get rid of last comma
  put randomList into msg
end mouseUp
```

Result

Syntax

```
the result
result()
```

Example

```
if the result is not empty then answer "Try again."
```

Description

The `result` function returns an explanatory text string if an immediately preceding `close file`, `convert`, `create stack`, `find`, `go`, `import paint`, `export paint`, `open file`, `picture`, `read`, or `save` command was unsuccessful. The `result` function returns empty if the command executed successfully. The value of `the result` can also be set by a `return` statement in a message handler or by an external command. The value of `the result` is reset by execution of another command and at the end of the handler.

Script

The following example handler searches for a string and displays either the string or the error message if it doesn't find the string:

```
on doMenu var
  if var is "Find..." then
    global findMe
    repeat
      ask "Find what string:" with findMe
      if It is not empty then find It
      else exit doMenu -- cancel clicked
        if the result is not empty then
          put the result into findMe -- display the error
        next repeat
      else
        put It into findMe -- otherwise display the string
        exit repeat
      end if
    end repeat
  else pass doMenu
end doMenu
```

Notes

It is safer to depend on the empty result of a successful execution, rather than the particular value of some error message because those values could be different in future versions of HyperCard.

If any of the commands listed in the `result` function description are sent from the Message box and generate an error, HyperCard displays the text of the `result` function in a dialog box. For example, if you sent a `go` command using the `without dialog` form from the Message box and used the name of a stack that doesn't exist, a dialog box containing "No such stack" is displayed after the command is sent. The same command sent in a handler would put the error message into `the result`, but would not display a dialog box. If you want a dialog box displayed, you could include an `answer` statement in your handler that displays `the result`.

For example

```
on mouseUp
  go stack quote& clickText &quote without dialog
  put the result into goError
  if goError is not empty
  then answer goError with "OK"
end mouseUp
```

Chapter 9 discusses the `return` statement. Appendix A contains information about external commands.

Round

Syntax

```
the round of factor
round(expression)
```

Factor and *expression* yield numbers.

Example

```
put round(resultVariable) into field 1
```

Description

The `round` function returns the source number rounded off to the nearest integer.

Any odd integer plus exactly 0.5 rounds up; any even integer (or 0) plus exactly 0.5 rounds down. If the source number is negative, HyperCard internally removes the negative sign, rounds its absolute value, then puts the negative sign back on.

Script

The following function handler rounds off an amount to the nearest dollar:

```
function roundToDollar amount
  set numberFormat to ".00" -- sets dollar format
  return round(amount)
end roundToDollar
```

ScreenRect

Syntax

```
the screenRect
screenRect()
```

Example

```
put (the screenRect) into menuLoc
```

Description

The `screenRect` function returns the rectangle of the screen in which the HyperCard menu bar is displayed; the value returned is four integers, separated by commas, representing the pixel offsets of the left, upper, right, and lower edges, respectively, from the upper-left corner of the screen.

Seconds

Syntax

```
the seconds
seconds()
```

Example

```
put (the seconds-startTime) into runTime
```

Description

The `seconds` function returns an integer showing the number of seconds between midnight, January 1, 1904, and the current time set in your Macintosh. The `seconds` function can be abbreviated `secs`.

Script

The following example handler counts the number of seconds the user holds down the mouse button:

```
on stopWatch
  put the long time into now -- what time is it now?
  convert now to seconds
  wait while the mouse is down
  -- wait until mouse is released
  put the seconds-now into msg
  -- how many seconds have elapsed?
end stopWatch
```

Note

See also the `convert` command in Chapter 10, "Commands."

SelectedChunk

Syntax

```
the selectedChunk
selectedChunk()
```

Examples

```
get selectedChunk
put the selectedChunk
```

Description

The `selectedChunk` function returns a chunk expression describing the location of the selected text or the insertion point in a card or background field. For selected text, it is in the form of first character selected to last character selected of the field containing the selected text; for example, `char 7 to 15 of card field 3`.

If no text is selected, but the insertion point is in a field, `selectedChunk` returns the number of the character on either side of the insertion point in reverse order with the highest number first. For example, if the insertion point is located between characters 9 and 10 in card field 2, `selectedChunk` returns `char 10 to 9 of card field 2`. If nothing is selected and the insertion point is not in a field, it returns `empty`.

Notes

If you use the `selectedChunk()` form, be sure to leave the parentheses empty or you will get an error message.

Changing the highlight state of a button (clicking a button with `autoHilite` set to `true`) deselects the text and causes the `selectedChunk` function to return empty. Many other actions, such as clicking the card, clicking a locked field, or running an `idle` message handler that periodically changes some part of the display, also deselect the text, so you should put the result of `selectedChunk` into a container before any other action takes place.

You can put the `selectedChunk` result into the Message box or other container from a script, but you get an empty result if you make a `selectedChunk` function call from the Message box. This is the correct result of the call since typing in the Message box and pressing the Return key or Enter key deselects any selected text.

You can get and set the text properties `textFont`, `textHeight`, `textSize`, and `textStyle` of the `selectedChunk`. See Chapter 12, "Properties" for more information about the text properties.

SelectedField

Syntax

```
the selectedField
selectedField()
```

Examples

```
get selectedField
put the selectedField
```

Description

The `selectedField` function returns the descriptor of the field containing the selected text or the insertion point.

Notes

If you use the `selectedField()` form, be sure to leave the parentheses empty or you will get an error message.

Changing the highlight state of a button (clicking a button with `autoHilite` set to `true`) deselects the field and causes the `selectedField` function to return empty. Many other actions, such as clicking the card or clicking a locked field, also deselect the text, so you should put the result of `selectedField` into a container before any other action takes place.

You can put the `selectedField` result into the Message box or other container from a script, but you get an empty result if you make a `selectedField` function call from the Message box. This is the correct result of the call because typing in the Message box and pressing the Return key or Enter key deselects any selected field.

SelectedLine

Syntax

```
the selectedLine
selectedLine()
```

Examples

```
get selectedLine
put the selectedLine
```

Description

The `selectedLine` function returns a chunk expression describing the line within a card or background field containing the selected text or insertion point. The `selectedLine` returns in the form

`line` *number* `of card field` *number*
`line` *number* `of bkgnd field` *number*

Notes

Lines in fields are defined as ending with the return character, and because text wraps in fields, there may be several lines of text on the screen between return characters. For example, line 8 would be the text between the seventh and eighth return characters.

If you use the `selectedLine()` form, be sure to leave the parentheses empty or you will get an error message.

Changing the highlight state of a button (clicking a button with `autoHilite` set to `true`) deselects the text and causes the `selectedLine` function to return empty. Many other actions, such as clicking the card or clicking a locked field, also deselect the text, so you should put the result of `selectedLine` into a container before any other action takes place.

You can put the `selectedLine` result into the Message box or other container from a script, but you get an empty result if you make a `selectedLine` function call from the Message box. This is the correct result of the call because typing in the Message box and pressing the Return key or Enter key deselects any selected text.

SelectedLoc

Syntax

```
the selectedLoc
selectedLoc()
```

Examples

```
get selectedLoc
put the selectedLoc
```

Description

The `selectedLoc` function returns a point at which the selected text begins. The `selectedLoc` returns the point as two comma-separated integers that represent the offset from the top-left corner of the card to the bottom (baseline) left of the selected text.

Note

The value returned by the `selectedLoc` function is the same as the value returned by the `TEGetPoint` routine, which is described in *Inside Macintosh,* Volume V.

SelectedText

Syntax

```
the selectedText
selectedText()
```

Examples

```
get selectedText
put the selectedText
```

Description

The `selectedText` function returns the selected text. For example, if the second word of line 1 of card field 1 was "fast" and you selected the second word in line 1 of card field 1, the `selectedText` function would return the text "fast".

Notes

`SelectedText` returns the same results as `put the selection` or `put the value of the selectedChunk`.

If you use the `selectedText()` form, be sure to leave the parentheses empty or you will get an error message.

Changing the highlight state of a button (clicking a button with `autoHilite` set to `true`) deselects the text and causes the `selectedText` function to return empty. Many other actions, such as clicking the card or clicking a locked field also deselect the text, so you should put the result of `selectedText` into a container before any other action takes place.

You can put the `selectedText` result into the Message box or other container from a script, but you get an empty result if you make a `selectedText` function call from the Message box. This is the correct result of the call because typing in the Message box and pressing the Return key or Enter key deselects any selected text.

ShiftKey

Syntax

```
the shiftKey
shiftKey()
```

Example

```
if the shiftKey is down then put ¬
numToChar(charToNum(msg)-32) into msg
```

Description

The `shiftKey` function returns the constant `up` if the Shift key is not pressed; `down` if it is pressed.

Note

See also the `commandKey` and `optionKey` functions earlier in this chapter.

Sin

Syntax

`the sin of` *factor*
`sin(`*expression*`)`

Factor and *expression* yield numbers.

Example

```
put the sin of 2 -- puts 0.909297 into the Message box
```

Description

The `sin` function returns the sine of the angle that is passed to it. The angle must be expressed in radians.

Note

Radians can be converted to degrees by multiplying by 180 and dividing the result by the value of the constant `pi`.

Sound

Syntax

```
the sound
sound()
```

Example

```
wait until the sound is "done"
```

Description

The `sound` function returns the name of the sound resource currently playing (such as `"boing"`) or the string `"done"` if no sound is currently playing. The sound function enables you to synchronize sounds with other actions, because scripts continue to run while sounds play.

Script

The following example handler repeats a series of visual effects until the sounds specified by the `play` command finishes:

```
on hotShot
  play "harpsichord" tempo 100 "e"
  repeat until the sound is "done"
    visual effect zoom open very fast to white
    go this card
  end repeat
end hotShot
```

Notes

The `"done"` string is returned as a literal; it's not a HyperTalk constant like `up` or `true`.

See also the `play` command in Chapter 10.

Sqrt

Syntax

```
the sqrt of factor
sqrt(expression)
```

Factor and *expression* yield numbers.

Example

```
put the sqrt of msg
-- converts the number in msg to its square root
```

Description

The `sqrt` function returns the square root of the positive number passed to it. If you pass a negative number, you get the result `-NAN(001)`, which means "not a number."

Stacks

Syntax

```
the stacks
stacks
```

Example

```
put the stacks into card field 2
```

Description

The `stacks` function returns a return-delimited list of the names of all the currently open stacks.

Tan

Syntax

```
the tan of factor
tan(expression)
```

Factor and *expression* yield numbers.

Example

```
put the tan of 2 -- puts -2.18504 into the Message box
```

Description

The `tan` function returns the tangent of the angle that is passed to it. The angle must be expressed in radians.

Note

Radians can be converted to degrees by multiplying by 180 and dividing the result by the value of the constant `pi`.

Target

Syntax

```
the target
target()
```

Example

```
if the target is "card id 2875" then pass mouseUp
```

Description

The `target` function returns a string indicating the original recipient of the message. The string returned is one of the following:

`stack "`*stackName*`"`
`bkgnd of card id` *number*
`card id` *number*
`bkgnd field id` *number*
`card field id` *number*
`bkgnd button id` *number*
`card button id` *number*

For example, the `target` function enables you to tell, in a `mouseUp` handler in a background, whether

- the mouse was clicked over a field or button (which either would have had no `mouseUp` handler or would have passed the message on explicitly); `the target` would return the button or field ID number
- the mouse was clicked outside the area of all buttons and fields; `the target` would return the card ID number
- the message was sent directly to the background with the `send` command; `the target` would return the background ID number

You can use `the target` in place of an object descriptor to determine any of the target's properties:

```
get the short name of the target
```

Note

The `send` command resets the value of the target to the value of the object the message is being sent to.

Ticks

Syntax

```
the ticks
ticks()
```

Example

```
put the ticks into clock
```

Description

The `ticks` function returns an integer representing the number of ticks (1/60 second) since the Macintosh was turned on or restarted.

Script

The following example handler measures how long it takes to go to the Help stack and find the word "ticks":

```
on mouseUp
  put the ticks into startTicks
  go help
  find "ticks"
  put (the ticks - startTicks) into howLong
  answer "It took" && howLong div 60¬
  && "second(s) to find Help."
end mouseUp
```

Time

Syntax

```
the [adjective] time
time()
```

Adjective can be `long`, `short`, or `abbreviated`, (or `abbrev`, or `abbr`).

Example

```
put the time into the Message box
```

Description

The `time` function returns the time as a text string. The `short` and `abbreviated` forms are the same, returning the hour and minutes, such as `8:55 AM`. The `long time` form returns seconds as well, such as `8:55:23 AM`.

Script

The following example records the time at which a field is updated:

```
on closeField
  put return & the time after card field "updateList"
end closeField
```

Note

An adjective can't be used to modify the form of the `time` function that uses parentheses. The time string returned by the `time` function can be in either 24-hour or A.M./P.M. format depending on the time format set in the Macintosh Control Panel.

Tool

Syntax

```
the tool
tool()
```

Example

```
if the tool is "field tool" then choose browse tool
```

Description

The `tool` function returns the name of the currently chosen tool. Possible values returned by the `tool` function are

```
browse tool      oval tool
brush tool       pencil tool
bucket tool      polygon tool
button tool      rectangle tool
curve tool       regular polygon tool
eraser tool      round rect tool
field tool       select tool
lasso tool       spray tool
line tool        text tool
```

Script

The following example handler chooses the proper tool to manipulate a button or field when you move the pointer over either object:

```
on mouseWithin -- works in card, background, or stack script
   if "button" is in the target and the optionKey is down
   then choose button tool
   else if "field" is in the target and the optionKey is down
   then choose field tool
end mouseWithin
```

Note

See also the `choose` command in Chapter 10.

Trunc

Syntax

```
the trunc of factor
trunc(expression)
```

Factor and *expression* yield numbers.

Example

```
put the trunc of someNumber into msg
```

Description

The `trunc` function returns the integer part of the number passed to it. Any fractional part is disregarded, regardless of sign.

Script

The following example handler draws rectangles in increasing sizes, using the `trunc` function to ensure that the computed values used with the `drag` command are integers:

```
on mouseUp
  push card
  doMenu "New Card"
  reset paint
  choose rectangle tool
  put 50 into left
  put 150 into right
  put 50 into top
  put 150 into bottom
  repeat 5 -- the drag command only takes integers
    drag from left,top to right,bottom
    put trunc(left/1.2) into left
    put trunc(right/1.2) into right
    put trunc(top/1.2) into top
    put trunc(bottom/1.2) into bottom
  end repeat
  choose browse tool
end mouseUp
```

Value

Syntax

```
the value of factor
value(expression)
```

Factor and *expression* yield any values.

Example

```
put the value of field "formula" into field "result"
```

Description

The `value` function evaluates the string derived from *factor* or *expression* as an expression.

Script

The following example handler demonstrates the `value` function by forcing a second level of evaluation of a variable:

```
on mouseUp
   put "3 + 4" into expression
   put expression -- yields "3 + 4"
   wait 2 seconds
   put value of expression -- yields 7
end mouseUp
```

Windows

Syntax

```
the windows
windows()
```

Examples

```
put the windows into It
-- puts the windows value into the variable It
windows()
put the windows into card field 1
```

Description

The `windows` function returns a return-delimited list containing the names of all of the windows currently available to HyperCard and the current stack. The windows include built-in palettes (Tool and Pattern), Fatbits, the Message box, open card windows, and external windows (for example, Message Watcher, navigator palette, Variable Watcher).

Note

The windows returned in the list may not currently be visible.

Chapter 12 **Properties**

This chapter describes HyperCard properties. Properties are the defining characteristics of objects, other elements such as menus and windows, and the HyperCard environment.

Object properties determine how objects look and act. Global properties control aspects of the overall HyperCard environment. Painting properties control aspects of the HyperCard painting environment, which is invoked when you choose a Paint tool. Window properties determine how card windows, the Message box, the Tools and Patterns palettes, and external windows are displayed. Menu and menu item properties control aspects of HyperCard menus. There are also a few properties that apply to the Message Watcher or the Variable Watcher.

This chapter includes a set of tables that list the HyperCard properties by category. Button properties, field properties, window properties, and so on, are each listed in a separate table. The tables are followed by complete descriptions of all the properties in alphabetical order.

Retrieving and setting properties

HyperTalk lets you retrieve most properties by using the property name as a function in a script or the Message box. You must precede the property name with the word `the` or follow it with `of` if it's an object or window property. You can't use parentheses after the property name, as you do with built-in functions. The following example retrieves the `location` property of button 1 and puts it into the Message box:

```
put the loc of button 1 into msg
```

You set properties with the `set` command:

```
set loc of button 1 to 100,100
```

Some properties can't be `set`, although other actions affect them. For example, the size of a stack can be changed by compacting it and by adding objects.

You can get the value of most properties with the `get` command:

`get the` *property* `of` *object*

Most of the examples in this chapter do not include the `get` form in the syntax, because it is faster to put the property directly into a container or variable, rather than using the `get` command and then the `put` command.

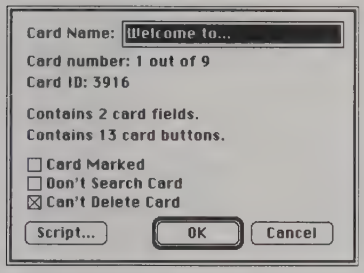
Object properties

You can see the value of many object properties by looking at an object's Info dialog box, an example of which is shown in Figure 12-1. (You bring up an object's Info dialog box by choosing the appropriate command from the Objects menu.)

You can also set many properties for the current object from the Info dialog boxes. To set the properties of any object in the current stack, you use the `set` command, either in a script or in the Message box.

Different HyperCard objects have different properties. For example, fields have a property determining their text style, but cards do not.

■ **Figure 12-1** An object's Info dialog box



Stack properties

Stack properties pertain to any stack on any disk or file server currently accessible to your Macintosh. A stack is specified as explained in Chapter 5, "Referring to Objects, Menus, and Windows." Settable properties of the current stack can be manipulated from a script or through the Stack Info dialog box invoked from the Objects menu.

The stack properties are listed in Table 12-1. More detailed information about each property is given later in this chapter.

■ **Table 12-1** Stack properties

Stack property name	Description
`cantAbort`	Controls whether or not the user can use Command-period to stop execution of scripts.
`cantDelete`	Controls whether or not the user can delete the specified stack.
`cantModify`	Controls whether or not the stack can be changed in any way.
`cantPeek`	Controls whether or not the user can look at button or field scripts with Command-Option.
`freeSize`	Determines the amount of free space of the specified stack in bytes.
`name`	Determines or changes the name of the specified stack, which is its Macintosh filename.
`script`	Retrieves or replaces the script of the specified stack.
`size`	Determines the size of the specified stack in bytes.

Background properties

Background properties pertain to any background in the current stack. The background is specified as explained in Chapter 5, "Referring to Objects, Menus, and Windows." Background properties can be manipulated from a script or from the Message box. Properties of the current background can also be manipulated through the Bkgnd Info dialog box invoked from the Objects menu.

The background properties are listed in Table 12-2. More detailed information about each property is given later in this chapter.

■ **Table 12-2** Background properties

Background property name	Description
`cantDelete`	Controls whether or not the user can delete the specified background.
`cantModify`	Controls whether or not the background can be changed in any way.
`dontSearch`	Determines whether or not the fields in a specified background are searched with the `find` command.
`ID`	Determines the permanent ID number of any background in the current stack.
`name`	Determines or changes the name of the specified background.
`number`	Determines the number of any background in the current stack.
`script`	Retrieves or replaces the script of the specified background.
`showPict`	Determines whether or not the picture of the specified background is shown.

Card properties

Card properties pertain to any card in the current stack. The card is specified as explained in Chapter 5, "Referring to Objects, Menus, and Windows." You can manipulate card properties from a script, in the Message box, or through the Card Info dialog box invoked from the Objects menu.

The card properties are listed in Table 12-3. More detailed information about each property is given later in this chapter.

■ **Table 12-3** Card properties

Card property name	Description
`cantDelete`	Controls whether or not the user can delete the specified card.
`cantModify`	Controls whether or not the card can be changed in any way.
`dontSearch`	Determines whether or not the fields in a specified card are searched with the `find` command.
`ID`	Determines the permanent ID number of any card in the current stack.
`marked`	Determines or changes whether a specified card is marked.
`name`	Determines or changes the name of the specified card.
`number`	Determines the number of any card in the current stack.
`rect[angle]`	Determines or changes the location and size of the rectangle occupied by the specified card window. See also Table 12-6.
`script`	Retrieves or replaces the script of the specified card.
`showPict`	Determines whether or not the picture of the specified card is shown.

Field properties

Field properties pertain to any card field or background field in the current stack. The field is specified as explained in Chapter 5, "Referring to Objects, Menus, and Windows." You can manipulate field properties from a script or from the Message box, or through the Field Info dialog box invoked from the Objects menu. (You must have the Field tool chosen and a specific card or background field selected to activate the Field Info dialog box.)

The field properties are listed in Table 12-4. More detailed information about each property is given later in this chapter.

■ **Table 12-4** Field properties

Field property name	Description
`autoTab`	Determines or changes whether or not the specified nonscrolling field sends the `tabKey` message to the current card.
`dontSearch`	Determines or changes whether or not the specified field is searched with the `find` command.
`fixedLineHeight`	Determines or changes whether or not the lines in the specified field have a fixed line height.
`ID`	Determines the permanent ID number of the specified field in the current stack.
`location`	Determines or changes the location of the specified field.
`lockText`	Determines or changes whether editing of text within the specified field in the current stack is allowed or prevented.
`name`	Determines or changes the name of the specified field.
`number`	Determines the number of the specified field in the current stack.
`rect[angle]`	Determines or changes the location and size of the rectangle occupied by the specified field. See also Table 12-6.
`script`	Retrieves or replaces the script of the specified field.
`scroll`	Determines or changes how much material is hidden above the top of the specified scrolling field's rectangle.
`sharedText`	Determines or changes whether the text in the specified background field appears on each card of that background.
`showLines`	Determines or changes whether the text baselines in the specified field appear or are invisible.
`style`	Determines or changes the style of the specified field in the current stack.
`textAlign`	Determines or changes the way lines of text are aligned in the specified field.
`textFont`	Determines or changes the font in which text in the specified field appears.

[continued]

■ **Table 12-4** Field properties [continued]

Field property name	Description
`textHeight`	Determines or changes the space between baselines of text in the specified field.
`textSize`	Determines or changes the type size in which text in the specified field appears.
`textStyle`	Determines or changes the style in which text in the specified field appears.
`visible`	Determines or changes whether the specified field is shown or hidden. The default value of the visible property is `true`.
`wideMargins`	Determines or changes whether some extra space is included at the left and right side of each line in the specified field (to make the text easier to read).

Button properties

Button properties pertain to any card button or background button in the current stack. The button is specified as explained in Chapter 5, "Referring to Objects, Menus, and Windows."

You can manipulate the properties of any button in the current stack from a script or from the Message box. Additionally, you can manipulate the properties of a button on the current card or background through the Button Info dialog box invoked from the Objects menu. (You must have the Button tool and a specific card or background button selected to activate the Button Info dialog.)

The button properties are listed in Table 12-5. More detailed information about each property is given later in this chapter.

■ **Table 12-5** Button properties

Button property name	Description
`autoHilite`	Determines or changes whether the specified button's `hilite` property is affected by the message `mouseDown`.
`hilite`	Determines or changes whether the specified button is highlighted (displayed in inverse video).
`icon`	Determines or changes the icon for the specified button in the current stack.
`ID`	Determines the permanent ID number of the specified button in the current stack.
`location`	Determines or changes the location of the specified button.
`name`	Determines or changes the name of the specified button.
`number`	Determines the number of the specified button in the current stack.
`rect[angle]`	Determines or changes the location and size of the rectangle occupied by the specified button. See also Table 12-6.
`script`	Retrieves or replaces the script of the specified button.
`sharedHilite`	Determines or changes whether the specified background button is displayed highlighted on all cards of that background.
`showName`	Determines or changes whether the name of the specified button (if it has one) is displayed in its rectangle on the screen.
`style`	Determines or changes the style of the specified button in the current stack.
`textAlign`	Determines or changes the way lines of text are aligned in the specified button.
`textFont`	Determines or changes the font in which text in the specified button appears.
`textHeight`	Determines or changes the space between baselines of text in the specified button.

[continued]

■ **Table 12-5** Button properties [continued]

Button property name	Description
`textSize`	Determines or changes the type size in which text in the specified button appears.
`textStyle`	Determines or changes the style in which text in the specified button appears.
`visible`	Determines or changes whether the specified button is shown or hidden. The default value of the visible property is `true`.

Rectangle properties

The properties described in this section pertain to the screen rectangles of buttons and fields, the Tools and Patterns palettes (called *windows* in scripts), the Scroll window, the Fatbits window, card windows, stack windows, the Message box, and external windows (if the XCMD that created the window supports control of rectangle properties).

The `rectangle` property itself also applies to buttons, fields, and windows. The `rectangle` property is two points, the top-left and bottom-right corners of an object's rectangle. The points are represented as four integers separated by commas: left (item 1), top (item 2), right (item 3), bottom (item 4). The rectangle properties affect these four items, one at a time or two at a time.

The rectangle properties are listed in Table 12-6. More detailed information about each property is given later in this chapter.

■ **Table 12-6** Rectangle properties

Rectangle property name	Description
`bottom`	Determines or changes the value of item 4 of the `rectangle` property when applied to the specified object or window.
`bottomRight`	Determines or changes items 3 and 4 of the value of the `rectangle` property when applied to the specified object or window.
`height`	Determines or changes the verticle distance in pixels occupied by the rectangle of the specified button or field; for windows, only determines the distance, can't change it.
`left`	Determines or changes the value of item 1 of the `rectangle` property when applied to the specified object or window.
`right`	Determines or changes the value of item 3 of the `rectangle` property when applied to the specified object or window.
`top`	Determines or changes the value of item 2 of the `rectangle` property when applied to the specified object or window.
`topLeft`	Determines or changes items 1 and 2 of the value of the `rectangle` property when applied to the specified object or window.
`width`	Determines or changes the horizontal distance in pixels occupied by the rectangle of the specified button or field; for windows, only determines the distance, can't change it.

Environmental properties

Some of the global properties, such as the `userLevel` property, can be set on the User Preferences card of the Home stack; others, such as the `lockMessages` property, can be retrieved and set only through HyperTalk. (However, the User Preferences card uses HyperTalk to set properties, and it could be extended to set any of the others.) The window properties, which pertain to the Message box and the tear-off palettes, can be set by clicking and dragging the windows themselves, as well as through HyperTalk. Painting properties, which pertain to the painting environment, can be controlled with the menus and palettes that appear when a Paint tool is selected, as well as through HyperTalk.

Global properties

You use global properties to choose how particular aspects of the HyperCard environment perform. You set global properties from any script or from the Message box, and their settings pertain to all objects—if you set `userLevel` to 3, for example, it remains 3 until you reset it (although a protected stack might impose some other user level while you are in that stack).

The global properties are listed in Table 12-7. More detailed information about each property is given later in this chapter.

■ **Table 12-7** Global properties

Global property name	Description
`blindTyping`	Determines or changes whether you can type messages into the Message box and send them (execute them) without having the Message box visible.
`cursor`	Determines or changes the image that appears at the pointer location on the screen.
`debugger`	Determines or changes the name of the current HyperTalk debugger. Custom debuggers can be created as XCMDs and called with HyperTalk.

[continued]

■ **Table 12-7** Global properties [continued]

Global property name	Description
`dragSpeed`	Determines or changes how many pixels per second the pointer moves when manipulated by all subsequent `drag` commands.
`editBkgnd`	Determines or changes where any painting or creating of buttons or fields happens—on the current card or on its background.
`language`	Determines or changes the language in which scripts are written and displayed.
`lockMessages`	Determines or changes whether HyperCard sends system messages such as `openCard`, `closeCard`, `newCard`, and `deleteCard`.
`lockRecent`	Determines or changes whether HyperCard adds miniature representations to the Recent card.
`lockScreen`	Determines or changes whether HyperCard updates the screen when you go to another card.
`longWindowTitles`	Determines or changes whether HyperCard displays the long name of a stack in its title bar.
`messageWatcher`	Determines or changes the name of the current message watcher. A custom message watcher can be created as an XCMD.
`numberFormat`	Determines or changes the precision with which the results of mathematical operations are displayed in fields and the Message box.
`powerKeys`	Determines or changes whether keyboard shortcuts for painting actions are available.
`printTextFont`	Specifies the text font to use when printing headers of print reports and variables.
`printTextStyle`	Specifies the text style to use when printing headers of print reports and variables.

[continued]

■ **Table 12-7** Global properties [continued]

Global property name	Description
`printTextSize`	Specifies the text size to use when printing headers of print reports and variables.
`printTextAlign`	Specifies the text alignment to use when printing headers of print reports and variables.
`printTextHeight`	Specifies the line height to use when printing headers of print reports and variables.
`printMargins`	Specifies the page margins to use when printing reports and expressions.
`scriptEditor`	Determines or changes the current script editor. A custom script editor can be created as an XCMD.
`scriptTextFont`	Determines or changes which font to use in the script editor.
`scriptTextSize`	Determines or changes which font size to use in the script editor.
`stacksInUse`	Determines which stacks are currently in the message-passing hierarchy.
`traceDelay`	Determines the time between execution of statements while tracing.
`textArrows`	Alters the function of the Right Arrow, Left Arrow, Up Arrow, and Down Arrow keys.
`userLevel`	Sets or retrieves the value of the current HyperCard user level.
`userModify`	Determines or changes whether or not a user can type into fields or use Paint tools in a stack that has been write-protected.
`variableWatcher`	Determines or changes the name of the current variable watcher. Custom variable watchers can be created as XCMDs.

Painting properties

Painting properties are aspects of the painting environment invoked when you choose a Paint tool from the Tools palette. Most of these properties are usually manipulated from the Options and Patterns menus that appear when a Paint tool is selected. The text attributes pertain to Paint text; they are usually manipulated from the dialog box that appears when you double-click the Paint Text tool in the Tools palette or when you choose Text Style from the Edit menu. Changes to the settings made from HyperTalk are reflected on their respective palettes and menus.

All of the painting properties can be restored to their default values simultaneously with the `reset paint` command, described in Chapter 10, "Commands."

The painting properties are listed in Table 12-8. More detailed information about each property is given later in this chapter.

■ **Table 12-8** Painting properties

Painting property name	Description
`brush`	Determines or changes the current brush shape used by the Brush tool.
`centered`	Determines or changes the Draw Centered setting.
`filled`	Determines or changes the Draw Filled setting.
`grid`	Determines or changes the painting Grid setting.
`lineSize`	Determines or changes the thickness of the lines drawn by the line and shape tools.
`multiple`	Determines or changes the Draw Multiple setting.
`multiSpace`	Determines or changes the amount of space left between edges of the multiple images drawn by the shape tools when the `multiple` property is `true`.
`pattern`	Determines or changes the current pattern used to fill shapes (including Paint text) and to paint with the Brush tool.

[continued]

■ **Table 12-8** Painting properties [continued]

Painting property name	Description
`polySides`	Determines or changes the number of sides of the polygon created by the Regular Polygon tool.
`textAlign`	Determines or changes the way characters are aligned around the insertion point as you type them with the Paint Text tool.
`textFont`	Determines or changes the font in which Paint text appears.
`textHeight`	Determines or changes the space between baselines of Paint text.
`textSize`	Determines or changes the font size in which Paint text appears on the screen.
`textStyle`	Determines or changes the style in which Paint text appears.

Window properties

Window properties let you find out about and change the way that the Message box, Tools palette, Patterns palette, card window, Scroll window, the navigator window, Message Watcher window, Variable Watcher window, stack window, and external windows are displayed. The window names these properties apply to are

```
message [box]                    pattern window or window "patterns"
message [window]                 tool window or window "tools"
msg                              card window
window "Scroll"                  window "navigator"
window "Message Watcher"         window "Variable Watcher"
window stackName                 window windowName
```

`Message` can be abbreviated `msg`.

The properties that apply only to the Message Watcher and Variable Watcher are listed in Table 12-11.

The window properties are listed in Table 12-9. If the property only applies to a specific window, it is called out in the description column. More detailed information about each property is given later in this chapter.

■ **Table 12-9** Window properties

Window property name	Description
`loc[ation]`	Determines or changes the location at which the window is displayed. The palette window location is relative to the current card window; the card window is reletive to the upper-left of the main screen.
`rect`	Determines the size of a window. See also Table 12-6. Does not apply to the Message Watcher window.
`rect[angle]`	Determines or changes the size of card and stack window rectangles. See also Table 12-6. Read-only property for the Message box.
`scroll`	Determines or changes the scroll of the specified card picture in the card window or picture in the specified picture window. Applies only to card windows that are smaller than the card picture.
`visible`	Determines or changes whether a window is shown or hidden on the screen.

Menu and menu item properties

The menu item properties described in this section pertain to any specified menu item created with the `put` command. You can manipulate the properties of menu items from a script or from the Message box.

You can use the `delete`, `disable`, and `enable` commands to delete, disable, or enable the Tools, Patterns, Font, and Apple menus, but you cannot alter the contents of those menus with any other HyperTalk commands. You can also enable and disable those menus with the `enabled` property and get the `name` property of those menus by referring to them by number. However, you cannot manipulate them with any of the other menu item properties.

The menu and menu item properties are listed in Table 12-10. More detailed information about each property is given later in this chapter.

■ **Table 12-10** Menu and menu item properties

Menu property name	Description
`checkMark`	Determines or changes the current checked value of a specified menu item; a Boolean value.
`cmdChar`	Determines or changes the current character to be used with the Command key as a keyboard shortcut for a specified menu item. The `cmdChar` property can be spelled out `commandChar`.
`enabled`	Determines or changes the enabled or disabled state of a specified menu or menu item; a Boolean value.
`markChar`	Determines or changes the current character that indicates a specified menu item is checked.
`menuMsg`	Determines or changes the current message to be sent when a specified menu item is choosen. The `menuMsg` property can also be spelled out `menuMessage`.
`name`	Determines or changes the specified menu or menu item name. This is a read-only property for the Tools, Patterns, Font, and Apple menus.
`textStyle`	Determines or changes the text style of the specified menu item.

Message Watcher and Variable Watcher properties

The Message Watcher and Variable Watcher properties described in this chapter pertain to the built-in Message Watcher or Variable Watcher external windows. You can manipulate their properties from a script or from the Message box.

The Message Watcher and Variable Watcher properties are listed in Table 12-11. More detailed information about each property is given later in this chapter.

■ **Table 12-11** Message Watcher and Variable Watcher properties

Property name	Description
Message Watcher	
`hideIdle`	Determines or changes whether the "Hide idle" check box is checked in the Message Watcher window.
`hideUnused`	Determines or changes whether the "Hide unused messages" check box is checked in the Message Watcher window.
Variable Watcher	
`hBarLoc`	Determines or changes the position of the horizontal bar in the Variable Watcher window.
`rect`	Determines or changes the size of the Variable Watcher window. Cannot be spelled out `rectangle` when it applies to the Variable Watcher window. Does not apply to the Message Watcher window.
`vBarLoc`	Determines or changes the position of the vertical bar in the Variable Watcher window.

HyperCard property descriptions

The rest of this chapter contains all of the HyperCard properties in alphabetical order for easy reference. The first line of each description tells which objects (stack, background, card, field, button, rectangle) or elements (menu item, Message Watcher, or Variable Watcher) or which environment (global, window, or painting) the property applies to.

Some of the syntax statements and examples in this chapter use the soft return (¬) character to continue long statements onto the next line. The soft return is used here because of the line-length limitations of the page format used in this chapter. You should avoid using soft returns in your scripts so that the statements in your handlers are easier to read.

To help you quickly locate properties that are new and enhanced since the first edition of this book, the names of the new and enhanced properties are followed with a symbol like the one at the beginning of this paragraph.

AutoHilite

Applies to Buttons

Syntax `set [the] autoHilite of` *button* `to` *boolean*

Button is an expression that yields a background or card button descriptor. *Boolean* is an expression that yields either `true` or `false`.

Example

```
set autoHilite of button 6 to true
```

Description You use the `autoHilite` property to determine or change whether the specified button's `hilite` property is affected when the button is pressed (that is, when the mouse button is held down while the mouse location is within the button rectangle.) The default value of `autoHilite` is `true` (button affected) for buttons created with New Button in the Objects menu and `false` for buttons created with the Command-drag combination.

When `autoHilite` is true, pressing the button sets the button's `hilite` property to `true`, and releasing the button sets its `hilite` property to `false`. The effect is that the button is momentarily highlighted (displayed in inverse video) when the user clicks it, giving visual feedback for the click action. The `mouseDown` message is sent to the button when the mouse is down and the `mouseUp` message is sent when the mouse is up.

Notes For radio or check box buttons that have the `autoHilite` property set to `true`, pressing the button changes the `hilite` property from `false` to `true` or `true` to `false`.

For transparent, opaque, rectangle, shadow, and round rectangle buttons that have the `autoHilite` property set to `true`, pressing the button sets the `hilite` property to `true` and releasing the button sets the `hilite` property to `false`.

The `mouseDown` or `mouseUp` messages by themselves do not affect the highlight state of a button. It is the action of pressing down and releasing the button that affects the highlight state when the `autoHilite` property is set to `true`.

The `autoHilite` property for a button can also be set to `true` or `false` by clicking the "Auto hilite" check box in the Button Info dialog box.

See also the `hilite` and `sharedHilite` properties, later in this chapter.

AutoTab

Applies to Fields

Syntax `set [the] autoTab of` *field* `to` *boolean*

Field is an expression that yields a nonscrolling background or card field descriptor. *Boolean* is an expression that yields either `true` or `false`.

Example

```
set autoTab of field 6 to true
```

Description You use the `autoTab` property to determine or change whether the specified nonscrolling field will send the `tabKey` message to the current card. When the `autoTab` property is `true`, pressing Return with the insertion point in the last line of that field moves the insertion point to the next field on that card.

The normal tabbing order for fields is as follows: if the field you are leaving is a card field, the insertion point moves to the next higher-numbered card field or the lowest-numbered background field if no higher-numbered card field exists; if the field you are leaving is a background field, the insertion point moves to the next higher-numbered background field or to the lowest-numbered card field if no higher-numbered background field exists.

Note The `autoTab` property can also be changed by clicking the "Auto tab" check box in the Field Info dialog box.

BlindTyping

Applies to Global environment

Syntax

```
set blindTyping to boolean
```

Boolean is an expression that yields either `true` or `false`.

Examples

```
set blindtyping to true
set blindtyping to false
put the blindtyping -- puts current value into message box
```

Description You use the `blindTyping` property to determine or change whether you can type messages into the Message box and send them (execute them) without having the Message box visible. Blind typing is available only if the user level is set to 5 (Scripting), and is usually set with a check box on the User Preferences card of the Home stack.

Notes Whether the default setting is `true` or `false` is determined at startup time and resume time by the setting chosen on the User Preferences card of the Home stack.

If you try to type into the Message box when it's hidden and `blindTyping` is `false`, HyperCard makes the computer beep.

Bottom

Applies to Buttons, fields, windows

Syntax `set [the] bottom of` *object* `to` *number*

Object is an expression that yields one of the following objects in the current stack:

a valid button descriptor
a valid field descriptor
`tool window` or `window "Tools"` (the Tools palette)
`pattern window` or `window "Patterns"` (the Patterns palette)
`window "Fatbits"`
`message [box]`
`message [window]`
`card window`
`window` *stackName*
`scroll window` or `window "scroll"`

Number is an expression that yields an integer representing the vertical offset in pixels from the top of the screen to the bottom of the specified object. *StackName* is an expression that yields the name of an open stack.

Examples

```
set bottom of button "Mover" to 64
put bottom of card button 4
put the bottom of this card window
set bottom of message box to 350
```

Description You use the `bottom` property to determine or change the value of item 4 of the `rectangle` property (left, top, right, bottom) when applied to the specified object or window.

Notes `Message` can be abbreviated `msg`.

See also the `rectangle` property, later in this chapter.

BottomRight

Applies to Buttons, fields, windows

Syntax `set [the] bottomRight of` *object* `to` *point*

Object is an expression that yields one of the following elements in the current stack:

a valid button descriptor
a valid field descriptor
`tool window` or `window "Tools"` (the Tools palette)
`pattern window` or `window "Patterns"` (the Patterns palette)
`window "Fatbits"`
`message [box]`
`message [window]`
`card window`
`window` *stackName*
`scroll window` or `window "scroll"`

Point is an expression that yields a list of two integers separated by a comma. *Point* represents the horizontal and vertical offsets, respectively, in pixels from the top-left corner of the card to the bottom-right corner of the specified object.

Examples

```
set bottomRight of bkgnd button id 23 to 64,100
put bottomRight of button "Mover"
put the bottomRight of window "Tools"
set bottomRight of message box to 250,30
```

Description You use the `bottomRight` property to determine or change the value of items 3 and 4 of the `rectangle` property (left, top, right, bottom) when applied to the specified object or window. The `bottomRight` property can be abbreviated `botRight`.

Notes `Message` can be abbreviated `msg`.

The Message Watcher and Variable Watcher windows do not have this property. See also the `rectangle` property, later in this chapter.

Brush

Applies to Painting environment

Syntax `set [the] brush to` *number*

Number is an expression that yields one of the numbers for the brush shapes shown in Figure 12-2.

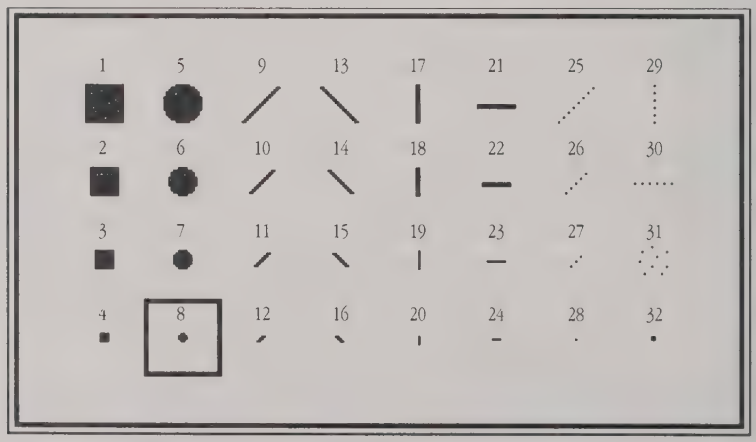
Example

```
set brush to 5
```

Description You use the `brush` property to determine or to change the current brush shape used by the Brush tool. It's normally manipulated from the Brush Shape dialog box invoked by choosing Brush Shape from the Options menu or by double-clicking the Brush tool.

The value of the `brush` property can be any integer from 1 to 32, each representing a brush shape from the Brush Shape dialog box. If you set the value of `brush` to a number lower than 1 or higher than 32, it automatically reverts to 1 or 32, respectively. The default `brush` setting is 7.

■ **Figure 12-2** Brush Shape dialog box and property values



CantAbort

Applies to Stacks

Syntax

```
set cantAbort of stack stackName to boolean
```

StackName is an expression that yields a stack name. *Boolean* is an expression that yields either `true` or `false`.

Examples

```
set cantAbort of this stack to true
set cantAbort of stack "Shoes and socks" to false
```

Description The `cantAbort` property pertains to any stack accessible to your Macintosh. It controls whether or not you can use Command-period to stop execution of a script. This property checks or unchecks the Can't Abort option in the Protect Stack dialog box.

Note The `cantAbort` property should be used with extreme caution. If a script fails in certain ways and `cantAbort` is true, your only option is to reboot.

Following is an example of an endless repeat loop:

```
put empty into fred
repeat
     add 1 to fred
     if fred < 0 the exit repeat
end repeat
```

If `cantAbort` is `true`, you must reboot to exit this loop.

CantDelete

Applies to Backgrounds, cards, stacks

Syntax `set cantDelete of` *object* `to` *boolean*

Object is an expression that yields a valid background, card, or stack descriptor. *Boolean* is an expression that yields either `true` or `false`.

Examples

```
set cantDelete of first card to true
set cantDelete of this bkgnd to true
set cantDelete of this stack to false
```

Description The `cantDelete` property pertains to any background, card, or stack accessible to your Macintosh. It controls whether or not the user can delete the specified object. The default value of `cantDelete` is `false`.

For backgrounds and cards, this property checks or unchecks the Can't delete option in the object Info dialog box of the specified object.

For stacks, this property checks or unchecks the Can't Delete Stack option in the Protect Stack dialog box. When the `cantDelete` property for a stack is set to `true`, the Delete Stack command in the File menu is unavailable. (If the user has checked Can't Delete Stack, however, and a script sets `cantModify` to `true` and then `false`, Can't Delete Stack is left checked.)

Note The `cantDelete` property is also automatically set when the user sets `cantModify`, as described in the next property description.

CantModify

Applies to Backgrounds, cards, stacks

Syntax

```
set cantModify of object to boolean
```

Object is an expression that yields a valid background, card, or stack descriptor. *Boolean* is an expression that yields either `true` or `false`.

Examples

```
set cantModify of this card to true
set cantModify of this stack to false
set cantModify of bkgnd "Yikes" to false
```

Description The `cantModify` property pertains to any background, card, or stack accessible to your Macintosh. It controls whether or not the background, card, or stack can be changed in any way.

For stacks, this property checks or unchecks the Can't Modify Stack option and the Can't Delete Stack option in the Protect Stack dialog box. When the `cantModify` property for a stack is set to `true`, the Compact Stack command in the File menu is unavailable. (If the user has checked Can't Delete Stack, however, and a script sets `cantModify` to `true` and then `false`, Can't Delete Stack is left checked.)

Notes When you set `cantModify` for a stack from a script, you override whatever the user last set manually in the Protect Stack dialog box. This works in the reverse as well. The user can overide the script by resetting the value in the Info dialog box. Setting `cantModify` to `false` does not, however, override protection provided by media that is write-protected in other ways.

See also the `cantDelete` and `userModify` properties in this chapter.

CantPeek

Applies to Stacks

Syntax `set cantPeek of stack` *stackName* `to` *boolean*

StackName is an expression that yields a stack name. *Boolean* is an expression that yields either `true` or `false`.

Examples

```
set cantPeek of this stack to true
set cantPeek of stack "TreeFrogs" to false
```

Description The `cantPeek` property pertains to any stack accessible to your Macintosh. It controls the ability to peek at scripts by pressing Command-Option-C (or B or S). It also controls whether or not you can view button outlines by pressing Command-Option. This property also checks or unchecks the Can't Peek option in the Protect Stack dialog box.

Centered

Applies to Painting environment

Syntax `set [the] centered to` *boolean*

Boolean is an expression that yields either `true` or `false`.

Examples

```
set centered to true
set centered to false
```

Description You use the `centered` property to determine or to change the Draw Centered setting. When `centered` is set to `true`, shapes are drawn from the center, rather than the corner. The default value of the `centered` property is `false`.

Note You can also set the `centered` property by choosing Draw Centered on the Options menu.

CheckMark

Applies to Menu items

Syntax

```
set [the] checkMark of menuItem of menu to boolean
```

Where *menuItem* is an expression that yields a menu item descriptor that is in the menu *menu*. *Menu* is an expression that yields a menu descriptor. *Boolean* is an expression that yields either `true` or false.

Examples

```
set checkMark of menuItem "Get Back" of menu "Direction" to false
put the checkMark of menuItem "Get Back" of menu "Direction"
if the checkMark of menuItem "Get Back" of menu "Direction"¬
is true then
set the enabled of menuItem "Get Back" of menu "Direction"¬
to false
```

Description You use the `checkMark` property to set or determine whether the check mark character for a menu item is currently displayed. When the `checkMark` property is set to `true`, a check mark character appears to the left of the menu item.

HyperCard does not automatically check or uncheck custom menu items each time a menu item is chosen, as it does for its own standard menus. Once an added menu item with a check mark is chosen, the check mark remains next to the menu item regardless of whether it is chosen again. You need to create handlers to keep track of the checked and unchecked menu items.

If you try to set or determine the `checkMark` property for a menu item that does not exist, HyperCard displays a "No such menu item" dialog box.

Notes See also the `create menu` and `put` commands in Chapter 10, "Commands," and the `markChar` property in this chapter.

CmdChar

Applies to Menu items

Syntax

```
set [the] cmdChar of menuItem of menu to char
```

Where *menuItem* is an expression that yields a menu item descriptor that is in the menu *menu*. *Menu* is an expression that yields a descriptor menu. *Char* is the character you want to use in combination with the Command key as the keyboard equivalent of the specified menu item.

Examples

```
set the cmdChar of menuItem "Get Back" of menu "Direction" to D
put the cmdChar of menuItem "Get Back" of menu "Direction"
```

Description You use the `cmdChar` property to set or determine the character to be used in combination with the Command key as the keyboard equivalent for a specified menu item. The character is displayed to the right side of the specified menu item along with the Command key symbol, ⌘.

Menu items usually send messages when invoked. The menu item message is specified with the `menuMsg` property, which is described later in this chapter.

`CmdChar` can also be spelled out as `commandChar`.

Notes If you try to set or determine the `cmdChar` property for a menu item that does not exist, HyperCard displays a "No such menu item" dialog box.

Command-key keyboard equivalents do not work for menu items that have been deleted or disabled. However, with a HyperCard standard menu item, you can still send a `doMenu` command to invoke a menu item action even when the menu item has been deleted.

See also the `menuMsg` property later in this chapter and the `create menu` and `put` commands in Chapter 10, "Commands."

Cursor

Applies to Global environment

Syntax

```
set cursor to cursorType
```

CursorType yields one of the cursor names or numbers listed in the description.

Examples

```
set cursor to 4
set cursor to plus
```

Description The `cursor` property determines the image that appears at the pointer location on the screen. The cursor setting is the ID number or name of a Macintosh 'CURS' resource, which must be available in the HyperCard file itself or in the current stack file. 'CURS' resources can be installed, removed, and created with an Macintosh resource editor.

HyperCard resets the cursor to the one for the current tool at idle, when no other action is happening. The available cursors all have a name and some of them have a number:

Number	Name
1	Ibeam
2	cross
3	plus
4	watch
	hand
	arrow
	busy
	none

Script The following handler spins the busy cursor two full revolutions. The busy cursor looks like a beach ball. Each time it is set, it turns 45° clockwise.

```
on spinMe
   repeat 16 times
      set the cusor to busy
      wait 4 ticks
   end repeat
end spinMe
```

Note You can't get the `cursor` property or use it as a function; you can only set it.

Debugger

Applies to Global environment

Syntax

```
set [the] debugger to debuggerName
```

DebuggerName is an expression that yields the name of the HyperCard debugger or a custom XCMD debugger.

Examples

```
set the debugger to "ScriptEditor"
set the debugger to "MyDebugger"
put the debugger
```

Description You use the `debugger` property to set or determine the current debugger to use when debugging HyperTalk scripts. The default HyperTalk debugger is a built-in XCMD named `ScriptEditor`.

If you set a name for the debugger that HyperCard can't find, it uses the built-in HyperTalk debugger.

Note See the description of the HyperTalk debugger in Chapter 3, and see Appendix A, "External Commands and Functions," for more information about custom debuggers.

DontSearch

Applies to Backgrounds, cards, fields

Syntax

```
set [the] dontSearch of object to boolean
```

Object is an expression that yields any valid background, card, or field descriptor. *Boolean* is an expression that yields either `true` or `false`.

Examples

```
set dontSearch of bkgnd 4 to true
put the dontSearch of bkgnd 3 into msg

if the name of bkgnd field 2 of this cd is "Secrets"
then
set dontSearch to true
```

Description You use the `dontSearch` property to set or determine whether or not the `find` command searches the specified background, card, or field in the current stack. When the `dontSearch` property of an object is set to `true`, the `find` command doesn't search that object. The default value for the `dontSearch` property is `false`.

When the `dontSearch` property of a background is set to `true`, the `find` command ignores all card or background fields on all of the cards of the specified background.

Note You can also set the `dontSearch` property by clicking the Don't Search check box in the object's Info dialog box.

DontWrap

Applies to Fields

Syntax `set [the] dontWrap of` *field* `to` *boolean*

Field is an expression that yields any valid field descriptor. *Boolean* is an expression that yields either `true` or `false`.

Example

```
set the dontWrap of bkgnd fld 4 to true
```

Description You use the `dontWrap` property to determine or change the `dontWrap` value for a field in the current stack. When the `dontWrap` property of a field is set to `true`, the text in the specified field does not wrap at the boundary of the field. The default value for the `dontWrap` property is `false` (wrap at the boundary of the field).

Note You can also set the `dontWrap` property by clicking the Don't Wrap check box in the Field Info dialog box.

DragSpeed

Applies to Global environment

Syntax `set dragSpeed to` *number*

Number is an expression that yields a positive integer; 1 is the slowest possible speed.

Example

```
set dragSpeed to 120
```

Description The `dragSpeed` property determines how many pixels per second the pointer moves when manipulated by all subsequent `drag` commands.

Script The following handler, placed in a button's script, creates a new card to draw on, sets a slow `dragSpeed` value, and slowly draws a diamond shape. It then fills the diamond with a pattern, waits a short time, deletes the card, and sends you back to the card where you started. You can change the `dragSpeed` property to drag faster by increasing the `dragSpeed` value, or slower by decreasing the value.

```
on mouseUp
  push card
  doMenu "New Card"
  set dragSpeed to 60
  choose line tool
  drag from 100,50 to 50,100
  drag from 50,100 to 100,150
  drag from 100,50 to 150,100
  drag from 100,150 to 150,100
  choose bucket tool
  set pattern to 10
  click at 100,100
  reset paint
  wait 25 ticks
  doMenu "Delete Card"
  choose browse tool
  pop card
end mouseUp
```

Note `dragSpeed` affects all of the Paint tools except the Bucket and Text tools. At idle time, HyperCard resets the `dragSpeed` property to 0. In this case, a value of 0 represents the fastest possible speed.

EditBkgnd

Applies to Global environment

Syntax `set editBkgnd to` *boolean*

Boolean is an expression that yields either `true` or `false`.

Examples

```
set editBkgnd to true
set editBkgnd to false
```

Description The `editBkgnd` property determines where any painting or creating of buttons or fields happens—on the current card (`false`) or on its background (`true`). It's usually set with the Edit menu and is available only when the user level is Painting (3) or higher. The default setting is `false` (editing on the card).

Enabled

Applies to Menus and menu items

Syntax

```
set [the] enabled of menuItem of menu to boolean
set [the] enabled of menu to boolean
```

Where *menuItem* is an expression that yields the descriptor of a menu item that is in the menu *menu*. *Menu* is an expression that yields a menu descriptor. *Boolean* is either `true` or `false`.

Examples

```
set enabled of menuItem 4 of menu "Utilities" to false
the enabled of menu "Home"
the enabled of menuItem "Repeat" of menu "Control"
put the enabled of the fifth menu
```

Description You use the `enabled` property to set or determine the state (either active or inactive) of a specified menu or menu item. When you create a menu or menu item, the default for `enabled` is `true` (active). If you set `enabled` to `false`, the menu item is dimmed (inactive).

If you try to set or determine the `enabled` property for a menu that does not exist, HyperCard displays a "No such menu" dialog box.

If you try to set or determine the `enabled` property for a menu item that does not exist, HyperCard displays a "No such menu item" dialog box.

Notes See also the `menuMsg` property in this chapter and the `create menu`, `disable`, `enable`, and `put` commands in Chapter 10, "Commands."

Command-key keyboard equivalents do not work on custom menu items that have been disabled.

Filled

Applies to Painting environment

Syntax `set [the] filled to` *boolean*

Boolean is an expression that yields either `true` or `false`

Examples

```
set filled to true
set filled to false
```

Description You use the `filled` property to determine or to change the Draw Filled setting. When `filled` is `true`, the current pattern in the Patterns palette is used to fill shapes as they are drawn. The default value of the `filled` property is `false`.

Note You can also set the `filled` property by choosing Draw Filled from the Options menu.

FixedLineHeight

Applies to Fields

Syntax

```
set [the] fixedLineHeight of field to boolean
```

Field is any expression that yields the descriptor of a field. *Boolean* is an expression that yields either `true` or `false`.

Examples

```
set the fixedLineHeight of field 6 to true
get fixedLineHeight of bknd field 3

if fixedLineHeight of field 6 is false then
put "fixedLineHeight is false"
```

Description You use the `fixedLineHeight` property to determine or specify whether a field has fixed line spacing when the text is of different sizes. If widely varying sizes of text are going to be used in a field, the value of the `fixedLineHeight` property needs to be set to `false`. The default setting of `fixedLineHeight` is `false`.

Notes You can also change the value of the `fixedLineHeight` property by clicking the Fixed Line Height check box in the Field Info dialog box.

The `fixedLineHeight` property is `true` for fields created with versions of HyperCard earlier than 2.0. When the same stacks are converted to the HyperCard 2.0 format, the `fixedLineHeight` becomes `false`.

The `fixedLineHeight` property is set to `true` when `showLines` is set to `true`. If `fixedLineHeight` property is set to `false`, the `showLines` is also set to `false`.

See the `textSize` property, later in this chapter, for more information on how to set different sizes of text in fields.

FreeSize

Applies to Stacks

Syntax `put [the] freeSize of stack` *stackName* `[into` *container*`]`

StackName is an expression that yields any stack name currently available to HyperCard and *container* is any field, variable, the selection, or the Message box.

Example

```
put freeSize of stack "dogfeathers" into field "Size"
```

Description You use the `freeSize` property to determine the amount of free space of the specified stack in bytes. (Free space changes in a stack each time you add or delete an object.)

Script The following handler compacts a stack based on a specified `freeSize` value.

```
on closeStack
 if the freeSize of this stack > 24000
 then doMenu "Compact Stack"
end closeStack
```

Note The `freeSize` property can be changed only by choosing Compact Stack from the File menu (or executing the HyperTalk command `doMenu Compact Stack`), which changes its value to 0.

Grid

Applies to Painting environment

Syntax `set [the] grid to` *boolean*

Boolean is an expression that yields either `true` or `false`.

Examples

```
set grid to true
set grid to false
```

Description You use the `grid` property to determine or to change the painting grid setting. When the value of `grid` is `true`, movement of many Paint tools is constrained to eight-pixel intervals. The default value of the `grid` property is `false`.

Note You can also set the `grid` property by choosing Grid from the Options menu.

HBarLoc

Applies to Variable watcher windows

Syntax

```
set [the] hBarLoc of window "Variable Watcher" to number
```

Number is an expression that yields a positive integer that represents the offset in pixels from the bottom of the Variable Watcher window title bar to the horizontal bar in the window.

Examples

```
set the hBarLoc of window "Variable Watcher" to 123

put the hBarLoc of window "Variable Watcher"
-- puts the hBarLoc value of the variable watcher in msg box
```

Description You use the `hBarLoc` property to determine or to change the current position of the horizontal bar in the Variable Watcher window. The horizontal bar determines how many of the variable name and value fields are visible in the Variable Watcher window.

Notes The built-in Variable Watcher window is a HyperCard XCMD. It can be replaced with a custom variable watcher XCMD by setting the `variableWatcher` property to the name of a variable watcher XCMD.

Custom variable watcher XCMDs can choose to respond to or ignore the `hBarLoc` property.

For more information about creating and calling a custom variable watcher XCMD, see Appendix A, "External Commands and Functions."

See also the description of the Variable Watcher in Chapter 3, "The Scripting Environment," and the `rect`, `variableWatcher`, and `vBarLoc` properties, later in this chapter.

Height

Applies to Buttons, fields, windows

Syntax `set [the] height of` *object* `to` *number*

Object is an expression that yields any valid button, field, or window descriptor. This is a read-only property for windows.

Number is an expression that yields a positive integer. The value *number* represents the total number of pixels in the vertical height of the specified object.

Examples

```
set the height of bkgnd button 2 to 60
put height of button 4
set the height of bkgnd field "phoneList" to 220
put the height of window "Home"
put the height of card window
```

Description You use the `height` property to determine or change the vertical distance in pixels occupied by the rectangle of the specified button or field. You can change the height of a button or field rectangle with the `set` command, but you can't set that property of any window including the Message box.

When you set the height of a button or field, its `location` property (center coordinate) remains the same.

Note See also the `rectangle` property, later in this chapter.

HideIdle

Applies to Message watcher windows

Syntax

```
set [the] hideIdle of window messageWindow to boolean
```

MessageWindow is an expression that yields the name of a message watcher window. *Boolean* is an expression that yields either `true` or `false`.

Examples

```
set the hideIdle of window "Message Watcher" to true
put hideIdle of window "Message Watcher"
```

Description You use the `hideIdle` property to determine or change whether the "Hide idle" checkbox is checked in a message watcher window.

The name of the message watcher window can be either the built-in HyperCard message watcher, `window "Message Watcher"`, or the name of a custom message watcher window that supports a "Hide idle" checkbox.

Note See also the description of the Message Watcher in Chapter 3, "The Scripting Environment." For more information about creating a custom message watcher XCMD, see Appendix A, "External Commands and Functions."

See also the `hideUnused` and the `messageWatcher` properties, later in this chapter.

HideUnused

Applies to Message watcher windows

Syntax `set [the] hideUnused of window` *messageWindow* `to` *boolean*

MessageWindow is an expression that yields the name of a message watcher window. *Boolean* is an expression that yields either `true` or `false`.

Examples

```
set the hideUnused of window "Message Watcher" to true
put hideUnused of window "Message Watcher"
```

Description You use the `hideUnused` property to determine or change whether the "Hide unused messages" checkbox is checked in the Message Watcher window.

The name of the message watcher window can be either the built-in HyperCard message watcher, `window "Message Watcher"`, or the name of a custom message watcher window that supports the "Hide unused messages" checkbox.

Note See also the description of the Message Watcher in Chapter 3, "The Scripting Environment." For more information about creating a custom message watcher XCMD, see Appendix A, "External Commands and Functions."

See also the `hideIdle` and the `messageWatcher` properties in this chapter.

Hilite

Applies to Buttons

Syntax `set [the] hilite of` *button* `to` *boolean*

Button is an expression that yields a background button or card button descriptor. *Boolean* is an expression that yields either `true` or `false`.

Examples

```
set hilite of button "You're on" to true
put the hilite of bkgnd button 3
```

Description You use the `hilite` property to determine or to change whether the specified button is highlighted (displayed in inverse video). The default value of the `hilite` property is `false`. To see what highlighting for the various button styles looks like, see the *HyperCard Reference*.

Notes The `hilite` property can be changed using the `set` command, either from a script or from the Message box, or, if the `autoHilite` property is `true`, by pressing the button. In that case, for all styles of buttons except check boxes and radio buttons, the `hilite` property becomes `true` when you press the button, and it becomes `false` when you release it.

For check boxes and radio buttons with their `autoHilite` property set to `true`, the `hilite` property toggles to its opposite state when the button is clicked and stays that way until it is clicked again. That is, when a check box is highlighted, it appears with an "X" check mark in its box; when it's not highlighted, the check mark does not appear. If `autoHilite` is `true`, an unselected check box displays an "X" when you click it; if you click it again, the "X" disappears. The appearance of the check mark correlates to the state of the button's `hilite` property. The situation is similar for radio buttons, except that the `true` highlighted state is indicated by a solid dot inside the button's circle.

See also the description of the `autoHilite` property, earlier in this chapter, and the `sharedHilite` property, later in this chapter.

Icon

Applies to Buttons

Syntax

```
set [the] icon of button to designator
```

Button is an expression that yields a background or card button descriptor. *Designator* yields the ID number of an available icon resource or the name of an icon (if it has one).

Examples

```
set icon of button "Instant" to 5005
set icon of button "Instant" to "DoGood"
put the icon of button "Instant"
```

Description You use the `icon` property to determine or to change the icon, if any, that is displayed with the specified button. If a button has no icon, the `icon` property is 0. An icon is identified by its ID number or by its name, if it has one.

Icons are small images that exist as Macintosh files and are editable with the HyperCard icon editor. For an icon to be displayed on a button, its resource must be available in the current stack, the Home stack, or the HyperCard application.

Notes The icon can also be changed by clicking the Icon button in the Button Info dialog box, which brings up another dialog box that displays the available icons graphically. When you click an icon in the icon display dialog box, the ID and name are displayed in the upper left-hand corner. The icon name is displayed in the dialog box only if the icon has a name.

If you use the `put` command with the `icon` property, you get the ID number of the icon, not the name. You cannot retrieve an icon name for a button.

ID

Applies to

Backgrounds, buttons, cards, fields

Syntax

```
put [the] [adjective] ID of object [into container]
```

A *adjective* is one of the `long`, `short`, and `abbreviated` modifiers as described in the section "Object ID Numbers" in Chapter 5. *Object* is an expression that yields any valid background, button, card, or field descriptor. C*ontainer* is the selection, any field, a variable, or the Message box.

Examples

```
put the long ID of bkgnd 3
if the ID of bkgnd 1 is 2282 then answer "Welcome Home"
put the short ID of card 35
put the id of field 1 into msg
put the ID of bkgnd button 3 into field "Button IDs"
put the short ID of card 35 after line 2 of field 2
```

Description

You use the `ID` property to determine the permanent ID number of any background, button, card, or field in the current stack.

You can't use the `set` command to change the ID of any object.

Script

The following script uses the `name` and `ID` properties to produce a list of button names and IDs. You need to create a field for the button name and ID list.

```
on mouseUp
  put empty into field "MyField"
  repeat with nums = 1 to the number of buttons
    put "Button name" && quote & short name of button nums &¬
    quote && "has id number" && id of button nums & return¬
    after field "MyField"
  end repeat
end mouseUp
```

You can place the script in any button or field, then click the button or field to get a list of all the buttons names and IDs on the current card. If you put the script into the same field you want to put the list, be sure to set the `lockText` property of that field to `true` so that the field receives the `mouseUp` message. Also, change all references to the field descriptor `field "MyField"` to match the descriptor for the field you created for the list. If the field created for the list is the same field the script is in, you can use `me` in place of `field "MyField"`.

Language

Applies to Global environment

Syntax `set language to` *languageName*

LanguageName is a text string that yields a language supported by your system.

Example

```
set language to French
```

Description You use the `language` property to choose the language in which scripts are written and displayed.

The languages available depend on the script translator resources available in your Home stack, stack, and HyperCard. The default setting is `English`, and it's always available.

Left

Applies to Buttons, fields, windows

Syntax `set [the] left of` *object* `to` *number*

Object yields one of the following elements in the current stack:

a valid button descriptor
a valid field descriptor
`tool window` or `window "Tools"` (the Tools palette)
`pattern window` or `window "Patterns"` (the Patterns palette)
`window "Fatbits"`
`message [box]`
`message [window]`
`card window`
`window` *stackName*
`scroll window` or `window "scroll"`

Number yields an integer that is the horizontal offset in pixels of the left side of the object. For objects visible on the screen, it is a positive integer between 1 and 319.

Examples

```
set left of button 2 to 34
put left of button 2
put the left of card field 3
set left of tool window to 65
```

Description You use the `left` property to determine or change the value of item 1 of the `rectangle` property (left, top, right, bottom) when applied to the specified object or window. The `left` property of an object can also be set to a value off the screen. Setting the `left` property of an object to a value off the screen makes the object seem as though it is hidden.

Notes `Message` can be abbreviated `msg`.

See also the `rectangle` property, later in this chapter.

LineSize

Applies to Painting environment

Syntax `set [the] lineSize to` *number*

Number yields a positive integer that is the total number of pixels in a line's width. It can be 1, 2, 3, 4, 6, or 8.

Example

```
set lineSize to 8
```

Description You use the `lineSize` property to determine or to change the thickness of the lines drawn by the line and shape tools. The default value of the `lineSize` property is 1. If you set the value of `lineSize` to a number lower than 1 or higher than 8, it automatically reverts to 1 or 8, respectively.

Note You can also set the `lineSize` property by choosing Line Size from the Options menu.

Location

Applies to Buttons, fields, windows

Syntax

```
set loc[ation] of object to point
```

Object is an expression that yields one of the following elements in the current stack:

a valid button descriptor
a valid field descriptor
`tool window` or `window "tools"` (the Tools palette)
`pattern window` or `window "patterns"` (the Patterns palette)
`message [box]`
`message [window]`
`card window`
`scroll window` or `window "scroll"`
`window "navigator"` (navigator palette)
`window` *stackName*
windows created with the `picture` command

Point is an expression that yields two integers separated by a comma. *StackName* is an expression that yields the name of a stack in quotation marks.

Examples

```
set loc of tool window to "100,100"
put the loc of field 3
put the loc of pattern window
set loc of msg to 30,150
set the loc of card window to 48,90
set the loc of window "navigator" to "45,60"
set the loc of scroll window to "165,45"
```

Description The `location` property sets or retrieves the location at which the window or object is displayed.

The point represents the horizontal and vertical offsets, respectively, in pixels from the top-left corner of the card to the center of a resizable window (a button or field) or the top-left corner of a nonresizable window (Tools palette, Patterns palette, Message box, the navigator palette, or Scroll window), disregarding the drag bar at the top of the window. The value for *point* must be in quotation marks for the navigator palette.

The point for a card window represents the horizontal and vertical offsets, respectively, in pixels from the top-left corner of the screen to the top-left corner of the card window.

The point for windows created with the `picture` command represents the horizontal and vertical offsets, respectively, in pixels from the top-left corner of the current card window to the top-left corner of the picture window. The value of *point* must be in quotation marks.

Notes

If you always put the value of *point* in quotation marks, it works with all of the HyperCard objects and elements for which you can set the location. An example that places quoted values in variables is shown under the `multiSpace` property, later in this chapter.

If you want to move a card on the screen, you set the `location` property for the card window of the current stack, not the location of the card.

The number that represents the horizontal offset for the card window is shifted to the closest multiple of 16 regardless of how you set it. For example, the statement `set the loc of cd window to 50,90` would result in the card location of `48,90`. It would shift to the next 16 pixels when the horizontal value of the `location` property reached the halfway point to the next higher or lower 16 pixels. For example, a horizontal value of `38` would shift the card window left to a horizontal offset of `32`.

When you move a card window with the `location` property, the system message `moveWindow` is sent. The `moveWindow` message is also sent when you drag the window to a new location, zoom it in or out with the zoom box, causing the `location` property to change, or show the card window at a new location with the `show` command.

See also the `rectangle` property later in this chapter, the `palette`, `picture`, and `show` commands in Chapter 10, "Commands," and the `moveWindow` system message in Chapter 8, "System Messages."

LockMessages

Applies to Global environment

Syntax `set lockMessages to` *boolean*

Boolean is an expression that yields either `true` or `false`.

Example

```
set lockMessages to true
```

Description You use the `lockMessages` property to prevent HyperCard from sending all `open`, `close`, `suspend`, and `resume` system messages. The default setting is `false`. HyperCard resets `lockMessages` to `false` at idle time (in effect, at the end of all pending handlers).

Note Setting the `lockMessages` property to `true` speeds up execution of scripts in which you go to cards, and those in which you create and delete objects. It also prevents execution of handlers invoked by system messages, which may be used to set up an environment—hiding the Message box, and so on. It's particularly useful when you want to go to a card to retrieve or write some information, but you don't want to stay there.

LockRecent

Applies to Global environment

Syntax `set lockRecent to` *boolean*

Boolean is an expression that yields either `true` or `false`.

Example

```
set lockRecent to true
```

Description You use the `lockRecent` property to prevent HyperCard from adding miniature representations to the Recent card. (The Recent card is invoked by Command-R or by choosing Recent from the Go menu).

The default setting is `false`. HyperCard resets `lockRecent` to `false` at idle time (in effect, at the end of all pending handlers).

Notes `LockRecent` is set to `true` when the `lockScreen` property is set to `true` regardless of the current setting of `lockrecent`. Setting the `lockRecent` property to `true` speeds up execution of scripts in which you go to cards.

See also the next property, `lockScreen`.

LockScreen

Applies to Global environment

Syntax `set lockScreen to` *boolean*

Boolean is an expression that yields either `true` or `false`.

Examples

```
set lockScreen to true
set lockScreen to false
```

Description You use the `lockScreen` property to prevent HyperCard from updating the screen when you go to another card.

The default setting is `false`. HyperCard resets `lockScreen` to `false` at idle time (in effect, at the end of all pending handlers).

Notes Setting the `lockScreen` property to `true` enables you to open different cards without displaying them on the screen, and it speeds up execution of scripts in which you go to cards. For example, you can lock the screen, then go to another card to read information out of a field, then return to the first card without having the second card appear to the user.

To ensure that the display is unlocked, each `set lockScreen to true` must be balanced with a `set lockScreen to false`.

See also the `lock screen` and `unlock screen` commands in Chapter 10.

LockText

Applies to Fields

Syntax `set [the] lockText of` *field* `to` *boolean*

Field is an expression that yields any valid card field or background field descriptor. *Boolean* is an expression that yields either `true` or `false`.

Examples

```
set the lockText of field "Glossary" to true
set the lockText of field "Glossary" to false
```

Description You use the `lockText` property to prevent or allow editing of text within a field in the current stack.

When the Browse tool is selected and the pointer is moved over an unlocked field, the pointer changes to an I-beam. Then, you can click the field and edit the text. If the field is locked (`lockText` is `true`), the cursor doesn't change, and the text cannot be edited. A locked field also receives the messages `mouseDown`, `mouseStillDown`, and `mouseUp` when you click it. The default value of `lockText` is `false`.

Notes You can also change this property by clicking the Lock Text check box in the Field Info dialog box.

When the `cantModify` property for the current stack is `true` and the `userModify` property is `false`, no changes can be made in a field. When `cantModify` is `true`, `userModify` is `true` and `lockText` is `false`, any editing done in a field is lost when the user moves to another card.

LongWindowTitles

Applies to Global environment

Syntax `set [the] longWindowTitles to` *boolean*

Boolean is an expression that yields either `true` or `false`.

Examples

```
set longWindowTitles to true
set the longWindowTitles to false
```

Description You use the `longWindowTitles` property to determine whether the stack name or the full pathname appears in the title bar of the card window.

The default value of `longWindowTitles` is `false`, show only the stack name in the title bar. If `longWindowTitles` is set to `true`, the full pathname of the stack is shown in the card window title bar.

Note See also Chapter 5, "Referring to Objects, Menus, and Windows," for more information about stack names and the full pathname of a stack.

MarkChar

Applies to Menu items

Syntax `set [the] markChar of` *menuItem* `of` *menu* `to` *char*

MenuItem is an expression that yields a menu item descriptor that is in the menu *menu*. *Menu* is an expression that yields a menu descriptor. *Char* is an expression that yields the check mark character you want to display in the menu to the left side of the specified menu item when it is chosen.

Examples

```
set markChar of menuItem "Blue" of menu "Colors" to >
put the markChar of menuItem 4 of the sixth menu
```

Description You use the `markChar` property to set or determine the check mark character for a menu item. Once a menu item is chosen, the check mark character appears to the left of the menu item.

HyperCard does not automatically check or uncheck custom menu items each time a menu item is chosen, as it does for its own standard menus. Once an added menu item with a check mark is chosen, the check mark remains next to the menu item regardless of whether it is chosen again. You need to create handlers to keep track of checked and unchecked custom menu items.

If you try to set or determine the `markChar` property of a menu item that does not exist HyperCard, displays a "No such menu item" dialog box.

Note See also the `put` command in Chapter 10, "Commands," and the `checkMark` property, earlier in this chapter.

Marked

Applies to Cards

Syntax

```
set [the] marked of card to boolean
```

Where *card* is an expression that yields the descriptor of any card within the current or specified stack. *Boolean* is an expression that yields either `true` or `false`.

Examples

```
put [the] marked of card 35 of stack "Ferns"
get the number of marked cards
set marked of card 2 to true
go to next marked card
print marked cards
show marked card
if the marked of this card is true then "doSomething"
```

Description You use the `marked` property to set or determine the marked state of any card in the current or specified stack. The default value of `marked` is `false`. A marked card or group of marked cards can be referred to in complicated searches and when printing.

Notes You can also set the marked property with the Card Marked option in the Card Info dialog box.

See also the `mark` and `unmark` commands in Chapter 10, "Commands."

MenuMsg

Applies to Menu items

Syntax

```
set [the] menuMsg of menuItem of menu to message
```

Where *menuItem* is an expression that yields a menu item descriptor that is in the menu *menu*. *Menu* is an expression that yields a menu descriptor. *Message* is an expression that yields a message to be sent by the menu item.

Examples

```
set the menuMsg of menuItem 7 of menu 10 to (doMenu "Prev")
put the menuMsg of menuItem "Next" of menu "Pictures"
put "Maple" after menu "Syrups" with menuMsg "go card 4"
```

Description You use the `menuMsg` property to specify the message sent to the current card by a specified menu item. You can also specify a menu message for a menu item when you create a menu item with the `put` command.

You may respond to the choosing of menu items in your scripts in two ways. Whenever a menu item is chosen, HyperCard sends a `doMenu` message. You can respond to the choosing of menu items by writing `doMenu` handlers in your scripts (this is the first way). If the `doMenu` message is not intercepted by any script, HyperCard checks the menu item's `menuMessage` property. If `menuMsg` is not empty, HyperCard sends that message to the current card. Therefore you can respond to the choosing of menu items by assigning them menu messages (this is the second way). If `menuMsg` is empty, HyperCard checks whether the item is one of its standard menu items. If it is a standard menu item, the standard behavior of that menu item occurs.

Custom menu items with the same name as a standard HyperCard menu item inherit the standard behavior of the HyperCard menu item. For example, if you add Background to a menu called Special, choosing it has the same effect as choosing the standard Background command from the Edit menu, unless you assign a custom menu message or intercept the `doMenu` message.

If you try to set or determine the `menuMsg` property of a menu item that does not exist, HyperCard displays a "No such menu item" dialog box.

Notes

In the case of the HyperCard standard menu items, the `doMenu` command works even when the item is deleted. For example, if the following handler is executed, HyperCard exits to the Finder:

```
on mouseDown
   delete menuItem "Quit HyperCard" from menu "File"
   doMenu "Quit HyperCard"
end mouseDown
```

The `menuMsg` property returns values for standard HyperCard menus only if they have been changed from the default HyperCard menu messages.

`MenuMsg` can be spelled out `menuMessage`.

You can't set the `menuMsg` property of menus items in the Tools, Patterns, Font, or Apple menus, except for the `menuMsg` property of the About HyperCard menu item in the Apple menu, which you can set.

See also the `create menu`, `delete`, `doMenu`, and `put` commands in Chapter 10 "Commands."

MessageWatcher

Applies to Global environment

Syntax

```
set [the] messageWatcher to name
```

Name is an expression that yields a valid message watcher XCMD name.

Examples

```
set messageWatcher to "MyWatcher"
put the messageWatcher -- puts the name of the current
                       -- message watcher in msg box
```

Description You use the `messageWatcher` property to determine or to change the current message watcher. The default value is `messageWatcher`, the built-in message watcher XCMD. You display the current message watcher with the `show` command or by setting the `visible` property of the message watcher window to `true`.

The built-in message watcher is a HyperCard XCMD. It can be replaced with a custom message watcher XCMD by setting the `messageWatcher` property to the name of a custom message watcher XCMD.

Notes See also the description of the Message Watcher in Chapter 3, "The Scripting Environment."

For more information about creating a custom message watcher XCMD, see Appendix A, "External Commands and Functions."

Multiple

Applies to Painting environment

Syntax `set [the] multiple to` *boolean*

Boolean is an expression that yields either `true` or `false`.

Examples

```
set multiple to true
set multiple to false
```

Description You use the `multiple` property to determine or to change the Draw Multiple setting. When `multiple` is `true`, multiple images are drawn as you drag a shape tool.

Tools affected by the multiple property are the line, rectangle, rounded rectangle, oval, and regular polygon tools. The default value of the `multiple` property is `false`.

Note You can also set the `multiple` property by choosing Draw Multiple from the Options menu. Setting `multiple` to `true` puts a check next to the Draw Multiple item in the Options menu. See also the `multiSpace` property, described next.

MultiSpace

Applies to Painting environment

Syntax `set [the] multiSpace to` *number*

Number is an expression that yields any positive integer.

Example

```
set multiSpace to 12
```

Description You use the `multiSpace` property to determine or to change the amount of space left between edges of the multiple images drawn by the shape tools when the `multiple` property is `true`. The default value of the `multiSpace` property is `1`.

Script The following script uses the `multiSpace` and `multiple` properties to create an interesting image. It also cleans up after it is completed.

```
on roundyRound
 doMenu "New Card"
 reset paint
 choose oval tool
 set multiple to true
 set dragSpeed to 200
 set multiSpace to 15
 put "60,25" into upperLeft
 put "260,175" into botRight
 put "260,25" into upperRight
 put "60,175" into botLeft
 drag from upperLeft to botRight with shiftKey
 drag from botRight to upperLeft with shiftKey
 drag from botLeft to upperRight with shiftKey
 drag from upperRight to botLeft with shiftKey
 wait 60 ticks
 set lockScreen to true
 choose browse tool
 play boing
 doMenu "Delete Card"
 doMenu "Back"
 set lockScreen to false
end roundyRound
```

Name

Applies to Backgrounds, buttons, cards, fields, stacks, menus, menu items

Syntax

```
set [the] name of object to objectName
set [the] name of menuItem of menu to itemName
[the] name of menuNumber
```

Object is an expression that yields a valid background, button, card, field, or stack descriptor. *ObjectName* is an expression that yields any valid HyperCard object name. The object name can be a maximum of 31 characters.

MenuItem is an expression that yields a menu item descriptor that is in the menu *menu*. *Menu* is an expression that yields a menu descriptor. *ItemName* is an expression that yields the new text to replace the current menu item name. *MenuNumber* is an expression that yields the number form of a menu descriptor.

Examples

```
set name of this stack to "TooHip"
put the long name of this stack into field 2
set name of this bkgnd to "TrueGrit"
put the long name of this background into field 3

put the short name of bkgnd 3 into word 3 of bkgnd field¬
"Background Name"

put the long name of field 3 into msg

if the name of card 1 of next bkgnd is "Begin"
then go home

set name of menuItem "Dogs" of menu "Animals" to "Canines"
put the name of the third menuItem of menu "Direction"
put the name of menu 8
put the name of the third menu
```

Description

You use the `name` property to determine or to change the name of the specified background, button, card, field, stack, or menu item. You can use the `name` property to determine the name of a menu, but not to set the name. A stack name must be a valid Macintosh filename.

When you use the `name of` *menuNumber* form of the `name` property to get the name of a menu, the descriptor for the menu is an expression that yields a valid number of one of the current HyperCard menus or custom menus in the menu bar. Menus are numbered from left ot right, starting with number 1 for the Apple menu.

Notes

If you try to retrieve an object's name when it has none, HyperCard returns its ID number.

If you try to modify or determine the `name` property of a menu item that does not exist, HyperCard displays a "No such menu item" dialog box.

If you try to modify or determine the `name` property of a menu that does not exist, HyperCard displays a "No such menu" dialog box.

You can set the name of the About HyperCard menu item in the Apple menu to a different value.

You can refer to menu items with the English names even after they have been localized. For example, the statement

```
set the name of menuItem "Next" of menu Go to "Next Card"
```

would work in Sweden.

See also the `put` command in Chapter 10, "Commands."

Number

Applies to Backgrounds, buttons, cards, fields

Syntax `put [the] number of` *object* `[into` *container*`]`

O*bject* is an expression that yields any valid background, button, card, or field descriptor. *Container* is the selection, any field, variable, or the Message box.

Examples

```
if the number of this bkgnd is 2
then
go next card

put number of last card into msg
```

Description You use the `number` property to determine the number of any background, button, card, or field in the current stack.

You can't set the number of a background, button, card, or field; the number of an object changes when you add or delete a background, a button, a card, or a field.

Note See also the `number` function in Chapter 11, which returns the count (how many) of various elements, not the descriptor number of an individual object.

NumberFormat

Applies to Global environment

Syntax `set numberFormat to` *formatType*

FormatType is an expression that yields the format (within quotation marks) that is to be used for the display of numbers.

Examples

```
set numberFormat to "00.##" -- display 02.21
set numberFormat to "0" -- display 2 for the same value
set numberFormat to "0." -- display 2.2, the default
```

Description The `numberFormat` property determines the precision with which the results of mathematical operations are displayed in fields and the Message box. Use zeros to show how many digits you want to appear, a period to show where you want the decimal point (if at all), and number signs (`#`) to the right of the decimal point in places where you want a trailing digit to appear, but only if it has value. Use zeros to the right of the decimal point if you always want the same number of digits to show, whether or not they have value. HyperTalk does calculations with up to 19 digits of accuracy.

HyperCard resets the `numberFormat` property to its default value, `"0.######"`, at idle time (in effect, at the end of all pending handlers).

Notes When you set the `numberFormat` property, you must enclose the value within double quotation marks if it contains a number sign (`#`).

The `numberFormat` property has no effect on how a number is displayed unless you perform a mathematical operation on it first. (For details, see Chapter 6, "Values".)

Pattern

Applies to Painting environment

Syntax `set [the] pattern to` *number*

Number is an expression that yields a positive integer in the range 1 to 32, each representing a pattern in the Patterns palette. The patterns are shown in Figure 12-3.

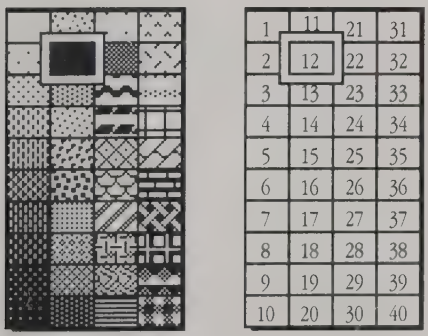
Examples

```
set pattern to 12
put the pattern
```

Description You use the `pattern` property to determine or to change the current pattern used to fill shapes (including Paint text) and to paint with the Brush tool.

The pattern numbers correspond to the 32 positions in the Patterns palette, not to a specific pattern.

■ **Figure 12-3** Patterns palette and pattern numbers



Notes You normally set the `pattern` property from the Patterns palette. You can edit a pattern by double-clicking a pattern on the Patterns palette.

PolySides

Applies to Painting environment

Syntax `set [the] polySides to` *number*

Number is an expression that yields a positive integer between 3 and 50. This integer is the number of sides in the polygon.

Example

```
set polySides to 12
```

Description You use the `polySides` property to determine or to change the number of sides of the polygon created by the Regular Polygon tool. The default value is `4`.

If you set the value of `polySides` to a number lower than 3 or higher than 50, it automatically reverts to 3 or 50, respectively. If you choose the circle in the Polygon Sides dialog box, the setting becomes 0.

Note You normally choose the Polygon Sides setting from a dialog box invoked by choosing Polygon Sides from the Options menu or by double-clicking the Regular Polygon tool.

PowerKeys

Applies to Global environment

Syntax `set powerKeys to` *boolean*

Boolean is an expression that yields either `true` or `false`.

Examples

```
set powerKeys to true
set powerKeys to false
```

Description You use the `powerKeys` property to provide keyboard shortcuts for painting actions. The availability of power keys is usually set on the User Preferences card of the Home stack. Power keys can be enabled only if the user level is set to Painting (3) or above.

The default setting is determined at startup and resume time by the setting on the User Preferences card of the Home stack.

Note Setting `powerKeys` to `true` in a script puts a check next to the Power Keys command in the Options menu and changes the setting on the User Preferences card.

PrintMargins

Applies to Global environment

Syntax `set [the] printMargins to` *rectangle*

Rectangle is an expression that yields four positive integers separated by commas. The integers are the left, upper, right, and lower sides, respectively, of a rectangle in pixels measured from the edges of the page.

Examples

```
set the printMargins to 78,78,340,440
the printMargins -- puts current margins in the msg box
```

Description You use the `printMargins` property to determine or change the current margin of the print area to be used when printing. The default value for the `printMargin` property is `0, 0, 0, 0`.

Note See also the `printTextAlign`, `printTextFont`, `printTextHeight`, `printTextSize`, and `printTextStyle` properties described in this chapter and the `open report printing` command in Chapter 10.

PrintTextAlign

Applies to Global environment

Syntax `set [the] printTextAlign to` *alignment*

Alignment is an expression that yields `right`, `left`, or `center`.

Examples

```
set the printTextAlign to Right
the printTextAlign -- puts the current alignment in the msg box
```

Description You use the `printTextAlign` property to determine or change the alignment used when printing the contents of a variable or when you want to modify the text alignment in the header of a print report job. The default value for the `printTextAlign` property is `left`.

Note See also the `printMargins`, `printTextFont`, `printTextHeight`, `printTextSize`, and `printTextStyle` properties described in this chapter and the `open report printing` command in Chapter 10.

PrintTextFont

Applies to Global environment

Syntax `set [the] printTextFont to` *font*

Font is an expression that yields a valid font name in the current Macintosh system or the name of a font in a font resource installed in HyperCard or in the current stack.

Examples

```
set the printTextFont to "Palatino"
the printTextFont -- puts the current font in the msg box
```

Description You use the `printTextFont` property to determine or change the font used when printing the contents of a variable or when you want to modify the font used in the header of a print report job. The default value for the `printTextFont` property is `Geneva`.

Note See also the `printMargins`, `printTextAlign`, `printTextHeight`, `printTextSize`, and `printTextStyle` properties described in this chapter and the `open report printing` command in Chapter 10.

PrintTextHeight

Applies to Global environment

Syntax `set [the] printTextHeight to` *number*

Number is an expression that yields a valid line height for a font in the current Macintosh system or a font in a font resource installed in HyperCard or in the current stack.

Examples

```
set the printTextHeight to 16
the printTextHeight -- puts current line height in the msg box
```

Description You use the `printTextHeight` property to determine or change the space between baselines of text when printing the contents of a variable or when you want to modify the line height of the text in the header of a print report job. The default value for the `printTextHeight` property is `13`.

Note See also the `printMargins`, `printTextAlign`, `printTextFont`, `printTextSize`, and `printTextStyle` properties described in this chapter and the `open report printing` command in Chapter 10.

PrintTextSize

Applies to Global environment

Syntax

```
set [the] printTextSize to number
```

Number is an expression that yields an integer that represents a valid font size in the current Macintosh system or the size of a font in a font resource installed in HyperCard or in the current stack.

Examples

```
set the printTextSize to 12
the printTextSize -- puts current text size in the msg box
```

Description You use the `printTextSize` property to determine or change the size of the font used when printing the contents of a variable or when you want to modify the size of the text in the header of a print report job. The default value for the `printTextSize` property is `10`.

Script This script sets some of the printing properties and prints the contents of card field 1, which is put in the variable `PJob`.

```
on printField
   put card field 1 into PJob
   set the printTextFont to "New York"
   set the printTextStyle to "Outline"
   set the printTextSize to "12"
   print PJob
end printField
```

Notes See also the `printMargins`, `printTextAlign`, `printTextFont`, `printTextHeight`, and `printTextStyle` properties described in this chapter and the `open report printing` command in Chapter 10.

PrintTextStyle

Applies to Global environment

Syntax `set [the] printTextStyle to` *style*

Style is an expression that yields a valid font style in the current Macintosh system or the style of a font in a font resource installed in HyperCard or in the current stack. Valid HyperCard font styles are `bold`, `condense`, `extend`, `italic`, `outline`, `plain`, and `underline`.

Examples

```
set the printTextStyle to Bold
the printTextStyle -- puts current style of the font in
                   -- the msg box
```

Description You use the `printTextStyle` property to determine or change the font style when printing the contents of a variable or when you want to modify the style of the text in the header of a print report job. The default value for the `printTextStyle` property is `plain`. The `group` style that is available in the HyperCard Style menu does not apply to printing.

Note See also the `printMargins`, `printTextAlign`, `printTextFont`, `printTextHeight`, and `printTextSize` properties described in this chapter and the `open report printing` command in Chapter 10.

Rect

Applies to Variable watcher windows, picture windows, stack windows

Syntax

```
set rect of window variableWatcher to location
set rect of window name to location
```

VariableWatcher is an expression that yields the name of a variable watcher window. *Location* is an expression that yields two points, reported as four positive integers separated by commas. *Name* is an expression that yields the name of a picture or stack window.

Examples

```
set rect of window "Variable Watcher" to "0,0,168,185"
put rect of window "MyWatcher" into msg
```

Description The `rect` property is two points, reported as four integers separated by commas. You use the `rect` property to determine or change the size of the variable watcher window.

The points represent the rectangle's upper-left (horizontal and vertical) and lower-right (horizontal and vertical) corner offsets in pixels, respectively, from the upper-left corner of the variable watcher window. The first point is always 0,0 and the second point is the offset from the first point.

Notes The `rect` property cannot be spelled out `rectangle` for the variable watcher window. It is a write-only property for the built-in Variable Watcher window.

`Rect` is also the abbreviated form of the `rectangle` property and works on all of the objects and windows that the `rectangle` property works on.

Properties that work on the HyperCard built-in external windows may not work on custom external windows. It is the responsibility of the creator of the custom window to provide support for HyperTalk external window properties. See also the `variableWatcher`, `hBarLoc`, and `vBarLoc` properties in this chapter.

Rectangle

Applies to Buttons, cards, fields, windows

Syntax

```
set [the] rect[angle] of object to location
```

Object is an expression that yields one of the following objects in the current stack:

a valid button descriptor
a valid field descriptor
a valid card descriptor
`tool window` or `window "tools"` (the Tools palette)
`pattern window` or `window "patterns"` (the Patterns palette)
`message [box]` or `message [window]`
`card window`
`window` *stackName*

Location is an expression that yields two points, reported as four positive integers separated by commas. *StackName* is the name of a stack.

Examples

```
set the rectangle of field 4 to 23,45,68,85
put rectangle of field "Sweet" into msg
put the rect of message box -- puts h,v,h,v into It
set rect of card window to 64,81,576,441
set the rect of this card to 0,0,512,360
```

Description The `rectangle` property is two points, reported as four integers separated by commas. You use the rectangle property to set or determine the size of buttons, fields, and windows. This property is a read-only property for the Message box, Tools palette, Patterns palette, and Scroll window.

The points represent the rectangle's upper-left (horizontal and vertical) and lower-right (horizontal and vertical) corner offsets in pixels, respectively, from the upper-left corner of the card.

You can set either of the rectangle points of a field or button beyond the boundaries of the card rectangle, putting the field or button out of view until you reset its coordinates through HyperTalk.

You can set the lower-right corner location of a button or field to a value smaller than the upper-left corner location, effectively causing the button or field to disappear. If you set a field to a size smaller than the minimum (12 by 12 pixels) but large enough to see, HyperCard resets it to the minimum size when you click it.

You can also change a button or field rectangle by dragging the upper-left or lower-right corner of the button or field with the appropriate tool selected (Button or Field).

Script

The following example handler, placed in a button script, is invoked when you click the button. It waits until you move the pointer outside the button rectangle, then beeps when you move the pointer back inside the button rectangle:

```
on mouseUp
 wait until the mouseLoc is not within rect of me
 repeat until the mouseLoc is within rect of me
  set cursor to busy -- spin beach ball while we wait
 end repeat
 beep
end mouseUp
```

Notes

The `rectangle` property can be abbreviated `rect`. The four integers that make up the rectangle property can also be changed individually and in various combinations. See the descriptions of `bottom`, `bottomRight`, `height`, `left`, `right`, `top`, `topLeft`, and `width` in this chapter. These are known collectively as *rectangle properties* and are summarized in Table 12-6.

The `rectangle` property does not affect palettes or the Message Watcher.

The operator `within` pertains to the rectangles of buttons and fields, the Tools and Patterns palettes, the Message box, and the screen on which the HyperCard menu bar is displayed. The syntax of an expression in which `within` is valid is the following:

location `is [not] within` *rectangle*

Location is an expression that yields a list of two integers separated by a comma, and *rectangle* is an expression that yields a list of four integers separated by commas.

When you resize a card window with the `rectangle` property a `sizeWindow` system message is sent. The `sizeWindow` message is also sent when you zoom a card window by clicking the zoom box or when you change its size with the scroll window (Command-E).

If the `location` property of a card window changes when you set the `rectangle` property, a `moveWindow` system message is sent. The `moveWindow` message is also sent when you drag the window to a new location, zoom it in or out, or show the window at a new location with the `show` command.

See also the `location` property, earlier in this chapter.

ReportTemplates

Applies to Stacks

Syntax `put [the] reportTemplates of stack` *stackName*

StackName is an expression that yields the name of a stack.

Examples

```
put reportTemplates of stack "People" into field 2
get the reportTemplates of stack "Forecast"
```

Description The `reportTemplate` property is a read-only property of stacks. It returns a return-delimited list of the names of the report templates for the specified stack.

Notes Report templates are created and saved for a stack with the Print Report command in the File menu. See the *HyperCard Reference* for more information about creating report templates.

See also the `open report printing` command in Chapter 10.

Right

Applies to

Buttons, fields, windows

Syntax

```
set [the] right of object to number
```

Object is an expression that yields one of the following objects in the current stack.

a valid button descriptor
a valid field descriptor
a valid card descriptor
`tool window` or `window "tools"` (the Tools palette)
`pattern window` or `window "patterns"` (the Patterns palette)
`message [box]` or `message [window]`
`card window`
`scroll window` or `window "scroll"`
`window` *stackName*

Number yields an integer that is the horizontal offset in pixels of the right side of the object.

Examples

```
set right of button 2 to 165
put right of button 2
put the right of the card window
set right of pattern window to 100
```

Description

You use the `right` property to determine or change the value of item 3 of the `rectangle` property (left, top, right, bottom) when applied to the specified object or window.

Note

See also the `rectangle` property, earlier in this chapter.

Script

Applies to Backgrounds, buttons, cards, fields, stacks

Syntax `set [the] script of` *object* `to` *scriptText*

O*bject* is the current background, button, card, field, or stack or any background, button, card, field, or stack name currently available to HyperCard. *ScriptText* yields any valid container that contains a script, or yields a text string that is a script.

Examples

```
set script of this stack to myScript & script of stack "TooHip"
set script of field "Effect" of first card to empty
set the script of second bkgnd to empty
set the script of third card to field 3
put the script of stack "home" into field "Home Script"
put the script of button "Train Set" into card field ¬
"last script"
```

Description You use the `script` property to retrieve or to replace the script of the specified object. The value of the `script` property is the text string composing the script of the specified stack.

When you set the `script` property using the `set` command, you replace it entirely.

Note Scripts are normally edited using the HyperCard script editor described in Chapter 3, "The Scripting Environment."

ScriptEditor

Applies to Global environment

Syntax

```
set [the] scriptEditor to name
```

Name is an expression that yields a valid script editor XCMD name.

Examples

```
set the scriptEditor to "MyEditor"
the scriptEditor -- puts current script editor in msg box
put the scriptEditor after field "Editor in Use"
```

Description You use the `scriptEditor` property to determine or change the current script editor. The default value for the `scriptEditor` property is `scriptEditor`, the name of the built-in script editor.

The built-in script editor is a HyperCard XCMD. It can be replaced with a custom script editor XCMD by setting the `scriptEditor` property to the name of a script editor XCMD.

Notes See also the `scriptTextFont`, `scriptTextSize`, and `scriptTextStyle` properties described later in this chapter and the description of the scripting environment in Chapter 3.

For more information about creating a custom script editor XCMD, see Appendix A, "External Commands and Functions."

ScriptTextFont

Applies to Global environment

Syntax

```
set [the] scriptTextFont to font
```

Font is an expression that yields a valid font name in the current Macintosh system or the name of a font in a font resource installed in HyperCard or in the current stack.

Examples

```
set the scriptTextFont to "Palatino"
the scriptTextFont -- puts current script editor font in
                   -- the msg box
```

Description You use the `scriptTextFont` property to determine or change the font in the script editor. The default value for the `scriptTextFont` property is `monaco`.

Notes See also the `scriptEditor`, `scriptTextSize`, and `scriptTextStyle` properties described in this chapter and the description of the scripting environment in Chapter 3.

ScriptTextSize

Applies to Global environment

Syntax `set [the] scriptTextSize to` *number*

Number is an expression that yields an integer that represents a valid font size in the current Macintosh system or the size of a font in a font resource installed in HyperCard or in the current stack.

Examples

```
set the scriptTextSize to 12
the scriptTextSize -- puts current script editor font in
                   -- the msg box
```

Description You use the `scriptTextSize` property to determine or change the size of the font used in the script editor. The default value for the `scriptTextSize` property is `9`.

Notes See also the `scriptEditor`, `scriptTextFont`, and `scriptTextStyle` properties described in this chapter and the description of the the scripting environment in Chapter 3.

ScriptTextStyle

Applies to Global environment

Syntax

```
set [the] scriptTextStyle to style
```

Style is an expression that yields a valid font style in the current Macintosh system or the style of a font in a font resource installed in HyperCard or in the current stack. The standard HyperCard font styles are `bold`, `condense`, `extend`, `italic`, `outline`, `plain`, and `underline`.

Examples

```
set the scriptTextStyle to Bold
the scriptTextStyle -- puts current style of the script editor
                    -- font in the msg box
```

Description You use the `scriptTextStyle` property to determine or change the current style of the font used in the script editor. The default value for the `scriptTextStyle` property is `plain`. The `group` style that is available in the HyperCard Style menu does not apply to the script editor.

Note See also the `scriptEditor`, `scriptTextFont`, and `scriptTextSize` properties described in this chapter and the description of the scripting environment in Chapter 3.

Scroll (windows)

Applies to Card windows, picture windows, stack windows

Syntax

```
set [the] scroll of [the] card window to point
set [the] scroll of window name to point
```

Point is an expression that yields two comma-separated positive integers that represent the point on the card or picture to be displayed at the upper-left corner of the window. *Name* is an expression that yields the name of a window created with the `picture` command.

Examples

```
set the scroll of the card window to 45,60
put the scroll of card window into scrollVar where scrollVar¬
-- is a variable name
set the scroll of window "Home" to "0,100"
```

Description You use the `scroll` property to determine or change the horizontal and vertical scroll (position) of the card window over the current card or the window over the current picture. This property allows you to scroll over a card that is larger than the area of the card window region. The default position of the `scroll` property for a card window is 0,0.

Notes The `scroll` property has no effect on card windows that are the same size as the card. Card windows can't be larger than the card. If the card is larger than the card window, you can enlarge a card window to show the entire card by setting the `rectangle` property for the card window, or use the Scroll window to view the card. You can also reset the value of the `scroll` property to reposition the card window with the Scroll command in the Go menu.

When setting the `scroll` property for a stack window, enclose the point parameters in quotation marks.

See also the `rectangle` property in this chapter and the `picture` command in Chapter 10.

Scroll (fields)

Applies to Fields

Syntax `set [the] scroll of` *scrollingField* `to` *number*

ScrollingField is any valid card or background scrolling field. *Number* is an expression that yields a positive integer representing the number of pixels that have scrolled above the top of the field rectangle.

Examples

```
set the scroll of field "Clues" to 0
put the scroll of field 1 div the textHeight of field 1¬
into linesAbove
```

Description You use the `scroll` property to determine or to change how much material is hidden above the top of a scrolling field's rectangle. Figure 12-4 illustrates the `scroll` property.

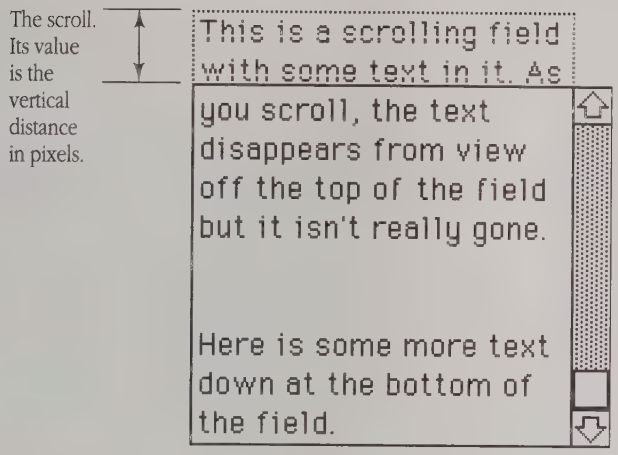
The value of the scroll property is `0` if the top of the field is visible. The number of text lines to which the `scroll` property correlates depends on the `textHeight` property of the field.

Notes You normally control how much material is above the top of the field rectangle by clicking or dragging in the scroll bar at the right side of the field.

If you try to get or set the `scroll` property of a nonscrolling field, you get an error dialog box.

The maximum scroll for a field is the farthest it can be scrolled using the scroll bar.

■ **Figure 12-4** The `scroll` property


The scroll. Its value is the vertical distance in pixels.
This is a scrolling field with some text in it. As you scroll, the text disappears from view off the top of the field but it isn't really gone.
Here is some more text down at the bottom of the field.

SharedHilite

Applies to Background buttons

Syntax `set [the] sharedHilite of` *button* `to` *boolean*

Button is an expression that yields any valid background button descriptor. *Boolean* is an expression that yields either `true` or `false`.

Examples

```
set sharedHilite of bkgnd button "Flip card" to true
put the sharedHilite of bkgnd button 2 into msg
```

Description You use the `sharedHilite` property to determine or to change whether the specified background button shares the same highlight state on all cards of that background. The default value for new background buttons is `false` (can have different highlight states on all cards of that background). When `sharedHilite` is set to `true`, the background button has the same highlight state on all cards of that background.

Notes Background buttons have two sets of highlight states, one you see when the button's `sharedHilite` is `true` and one you see when its `sharedHilite` is `false`. If you have a background button with its `sharedHilite` set to `false`, that button on each of those cards of that background can have a different highlight state (determined by the `hilite` property). If you change the `sharedHilite` property to `true` on that background button, the highlight state of that button on all of the cards of that background is set to `false` (not highlighted). The unshared highlight states of that background button are not lost, however. The unshared highlight states are stored separately with each card and can be returned to their previous values by setting the `sharedHilite` property back to `false`.

Background buttons copied and pasted to other cards have the same `sharedHilite` value as the button originally copied.

You can also change the `sharedHilite` property by clicking the Shared Hilite check box in the Button Info dialog box. See also the descriptions of the `autoHilite` and `hilite` properties, earlier in this chapter.

SharedText

Applies to Background fields

Syntax `set [the] sharedText of` *field* `to` *boolean*

Field is an expression that yields any valid background field descriptor. *Boolean* is an expression that yields either `true` or `false`.

Example

```
set the sharedText of card field 3 to true
```

Description You use the `sharedText` property to determine or to change whether the text in the specified background field appears on each card of that background. When the value of `sharedText` is `true`, the text in the specified background field is shared by all cards of that background. When it is `false`, the text can be different in the specified field on all the cards of that background. The default value of the `sharedText` property for new background fields is `false`.

Notes If you change the `sharedText` property to `true` on a background field that previously had the `sharedText` property set to `false` (unshared text), no text is displayed in that field on any of the cards of that background. The previous unshared text of that background field on each of the cards with that background is not lost, however. The unshared text is stored separately with each card and can be redisplayed in the background field of those cards with that background by setting the `sharedText` property back to `false`.

If you have a background field that contains shared text and you want to remove the text and change the field so that the text is unshared, you can cut the background field and paste it right back. It will be located in the same place, the text will be deleted, and the background field's `sharedText` property will be set to `false`. The ID of that background field will, however, be changed. So if any of your scirpts use that background field's ID, you need to change the scripts to reflect the new ID.

You can also change `sharedText` by clicking the Shared Text check box in the Field Info dialog box.

ShowLines

Applies to Background fields

Syntax `set [the] showLines of` *field* `to` *boolean*

Field is an expression that yields any valid field descriptor. *Boolean* is an expression that yields either `true` or `false`.

Examples

```
set the showLines of field four to true
put the showLines of card field 1 into msg
```

Description You use the `showLines` property to determine or to change whether the text baselines in the card or background field show or not. The default value of the `showLines` property is `false` (lines don't show).

Notes You can also change `showLines` by clicking the Show Lines check box in the Field Info dialog box.

ShowName

Applies to Buttons

Syntax `set [the] showName of` *button* `to` *boolean*

Button is an expression that yields any valid background or card button descriptor. *Boolean* is an expression that yields either `true` or `false`.

Examples

```
set showName of button "You who" to true
put the showName of button "You who" into msg
```

Description You use the `showName` property to determine or to change whether the name of the specified button (if it has one) is displayed in its rectangle on the screen. The default value is `false` (not displayed).

Note You can also change this property by clicking the Show Name check box in the Button Info dialog box.

ShowPict

Applies to Cards, backgrounds

Syntax

```
set [the] showPict of object to boolean
```

Object is an expression that yields any valid background or card descriptor. *Boolean* is an expression that yields either `true` or `false`.

Examples

```
set showPict of next card to false
set the showPict of this bkgnd to false
put showPict of bkgnd 3
```

Description You use the `showPict` property to determine or to change whether the picture on the specified card or background (if it has one) is displayed on the screen. The default value is `true` (displayed).

Setting the `showPict` property of a card to `false` is the same as hiding it with the `picture` form of the `hide` command. Setting it to `true` is the same as showing it with the `picture` form of the `show` command.

Notes When the `showPict` property of the current background or card is `false` and you try to use the Paint tool on it manually, a dialog box appears asking if you want to make the picture visible; clicking OK sets the `showPict` property to `true` and the picture appears. You can draw on hidden pictures from a script.

See also the `show` and `hide` commands in Chapter 10.

Size

Applies to Stacks

Syntax

```
put [the] size of stack stackName [into container]
```

StackName is the current stack or an expression that yields any stack name currently available to HyperCard. *Container* is an expression that yields any valid container.

Examples

```
get size of stack "Home"
put size of stack "Home" into field "Home Size"
```

Description You use the `size` property to determine the size of the specified stack in bytes.

Script The following handler examines a stack to see if it fits on an 800 KB disk.

```
on closeStack
 if size of this stack > 795000
 answer "This stack won't fit on an 800 KB disk"
end closeStack
```

Notes The minimum stack size is 4096 bytes; the theoretical maximum is 512 megabytes.

The `size` property can't be changed with the `set` command; it's changed only by adding things to and deleting things from the stack. (You must then compact the stack for the deletions to affect its size.)

StacksInUse

Applies to Global environment

Syntax `put [the] stacksInUse [into` *container*`]`

Container is an expression that yields any valid container.

Examples

```
the stacksInUse -- puts stack list in msg box
put the stacksInUse into field "Message Path"
```

Description You use the `stacksInUse` property to determine the stacks that have been added to the current message-passing hierarchy. `StacksInUse` returns a return-delimited list of the stacks in the current message-passing hierarchy. The stacks are listed in the order in which they are currently placed in the hierarchy.

The `stacksInUse` property can't be changed with the `set` command; the stacks in the current message-passing hierarchy can only be changed with the `start using` and `stop using` commands.

Notes See also the `start using` and `stop using` commands in Chapter 10 and the description of the message-passing hierarchy in Chapter 4.

If the message-passing hierarchy hasn't been altered with the `start using` command, `stacksInUse` returns empty.

Style

Applies to Buttons, fields

Syntax `set [the] style of` *object* `to` *objectStyle*

Object is an expression that yields any valid button or field descriptor. *ObjectStyle* is an expression that yields one of the valid field or button styles. Button styles are `transparent`, `opaque`, `rectangle`, `roundRect`, `checkBox`, or `radioButton`. Field styles are `transparent`, `opaque`, `rectangle`, `shadow`, or `scrolling`.

Examples

```
set the style of field 1 to scrolling
set style of button "You who" to roundRect
put the style of button 3 into msg
set the style of field 2 of card 4 to transparent
```

Description You use the `style` property to determine or to change the style of any button or field in the current stack.

Note You can also change the button or field style by clicking one of the Style radio buttons in the Button or Field Info dialog box. Some useful peculiarities of radio buttons and check box buttons are described under the `hilite` property, in this chapter.

TextAlign

Applies to Buttons, fields, painting environment

Syntax

```
set [the] textAlign [of object] to alignment
```

Object is an expression that yields a button or field descriptor. *Alignment* is an expression that yields one of the words `left`, `right`, or `center`.

Examples

```
set the textAlign of field 1 to left
set textAlign to center -- for paint text
```

Description You use the `textAlign` property to determine or to change the way characters are aligned around the insertion point as you type them. This property applies to Paint text, button name text, and the text in fields. The default value of the `textAlign` property is `left`.

Notes For Paint text, you can also set the `textAlign` property from the Font dialog box, which is invoked by choosing Text Style from the Edit menu, by double-clicking the Paint Text tool, or by pressing Command-T.

For buttons or fields, you can also set the `textAlign` property by choosing one of the text alignment options from the Font dialog box. To invoke the Font dialog box, you click the Font button in the Button or Field Info dialog box, choose Text Style from the Edit menu, or press Command-T when a button or field is selected.

See also the printTextAlign property, earlier in this chapter.

TextArrows

Applies to Global environment

Syntax `set textArrows to` *boolean*

Boolean is an expression that yields either `true` or `false`.

Examples

```
set textArrows to true
set textArrows to false
```

Description The `textArrows` property alters the function of the Right Arrow, Left Arrow, Up Arrow, and Down Arrow keys.

The default value of the `textArrows` property is `false`. When the `textArrows` property is `false`, the Right Arrow and Left Arrow keys take you to the next and previous cards in the stack, respectively, and the Up Arrow and Down Arrow keys take you forward and backward, respectively, through the cards you've already viewed.

When the `textArrows` property is `true`, the arrow keys move the text insertion point around in a field that you've opened for text editing or in the Message box if you've clicked in it. In the Message box, the Up Arrow and Down Arrow keys move the insertion point to the beginning and end of the line of text, respectively.

Note When the `textArrows` property is `true`, holding down the Option key while you press the arrow keys produces the same effect as pressing them alone when `textArrows` is `false`.

TextFont

Applies to Buttons, fields, painting environment

Syntax

```
set [the] textFont [of chunk] of field to font
set [the] textFont [of object] to font
```

Chunk is any valid chunk expression. *Field* is an expression that yields a field descriptor. *Object* is an expression that yields a button or field descriptor. *Font* is an expression that yields one of the font names available in your Macintosh system or in the font resources in HyperCard, the Home stack, or the current stack.

Examples

```
set textFont of field 1 to "courier"
set the textFont of bkgnd button 3 to helvetica
set textFont to Palatino -- for paint text
```

Description You use the `textFont` property to determine or to change the font in which text appears. This property applies to button text, field text, and Paint text. The default value of the `textFont` property is `geneva`.

Notes For Paint text, you can also set the `textFont` property from the Font dialog box, which is invoked by choosing Text Style from the Edit menu, by double-clicking the Paint Text tool, or by pressing Command-T while using a Paint tool.

For buttons or fields, you can also set this property by choosing one of the font names from the Font menu or from the Font dialog box. To invoke the Font dialog box, you click the Font button in the Field or Button Info dialog box, choose Text Style from the Edit menu, or press Command-T while a button or field is selected.

If you reset the default font for a field with the `textFont` property, any text that is already in that field is updated. If you try to set the `textFont` property to a font that doesn't exist, HyperCard sets it to `geneva`.

If different fonts are in a text selection, `textFont` returns the result `mixed`.

See also the `selectedChunk`, and `selectedLine` functions in Chapter 11, and the `printTextFont` and `ScriptTextFont` properties, earlier in this chapter.

TextHeight

Applies to Buttons, fields, painting environment

Syntax `set [the] textHeight [of` *object*`] to` *number*

Object is an expression that yields a button or field descriptor. *Number* is an expression that yields any positive integer.

Examples

```
set textHeight of field 1 to 20
set textHeight to 20 -- for paint text
```

Description You use the `textHeight` property to determine or to change the space between baselines of button text, field text, and Paint text. The value of the `textHeight` property is in pixels.

Notes For Paint text, you can also set the `textHeight` property in the Line Height box in the Font dialog box, which is invoked by choosing Text Style from the Edit menu, by double-clicking the Paint Text tool, or by pressing Command-T.

For buttons or fields, you can also set this property by typing the line height in the Line Height box in the Font dialog box. To invoke the Font dialog box, you click the Font button in the Field or Button Info dialog box, choose Text Style from the Edit menu, or press Command-T when a button or field is selected.

Although you can set this property for a button, it is meaningless because button-name text has only one line. See also the `fixedLineHeight`, `printTextHeight`, and `scriptTextHeight` properties described earlier in this chapter.

TextSize

Applies to Buttons, fields, painting environment

Syntax

```
set [the] textSize [of chunk] of field to number
set [the] textSize [of object] to number
```

Chunk is any valid chunk expression. *Field* is an expression that yields a field descriptor. *Object* is an expression that yields a button or field descriptor. *Number* is an expression that yields any positive integer.

Examples

```
set textSize of field 1 to 18
set the textSize of word 3 of line 4 to 12
set textSize to 18 -- for paint text
```

Description You use the `textSize` property to determine or to change the font size in which text appears on the screen. The `textSize` property applies to button text, text in fields, and Paint text. The value of the `textSize` property is in points. The default value of the `textSize` property is 8.

Although you can use any integer for `textSize`, exact sizes of fonts available look best. Fonts available are in the Macintosh system or in the font resources in your stack or HyperCard.

Notes For Paint text, you can also set the `textSize` property from the Font dialog box, which is invoked by choosing Text Style from the Edit menu, by double-clicking the Paint Text tool, or by pressing Command-T while using a Paint tool.

For buttons or fields, you can also set this property from the Style menu or by selecting one of the font sizes shown or typing directly in the Size box in the Font dialog box. To invoke the Font dialog box, you click the Font button in the Field or Button Info dialog box, choose Text Style from the Edit menu, or press Command-T while a button or field is selected.

If you reset the default text size for a field with the `textSize` property, any text that is already in that field is updated.

If different sizes of text are in a text selection, `textSize` returns the result `mixed`.

See also the `printTextSize` and `scriptTextSize` properties, earlier in this chapter.

TextStyle

Applies to Buttons, fields, painting environment

Syntax

```
set [the] textStyle [of chunk] of field to style
set [the] textStyle [of object] to style
```

Chunk is any valid chunk expression. *Field* is an expression that yields a field descriptor. *Object* is an expression that yields a button or field descriptor. *Style* is is an expression that yields a value of `plain` or any combination of the following: `bold`, `italic`, `underline`, `outline`, `shadow`, `condense`, `extend`, and `group` (separated by commas).

Examples

```
set textStyle to plain -- for paint text
set textStyle to bold,italic,underline -- for paint text
set textStyle of field 1 to plain
set the textStyle of line 1 of field 1 to bold,group
set the textStyle of the first card field to bold
```

Description You use the `textStyle` property to determine or to change the style in which text appears. The `textStyle` property applies to button text, text in fields, and Paint text. Its default value is `plain`. If you use `plain` in combination with any of the other values, the other values override `plain`.

Notes You use the `group` text style to group characters, words, or lines together so they are seen as a unit by HyperTalk. The `group` style does not apply to Paint text or button text.

Group text is supported through the `mouseDown` and `mouseUp` messages that are sent to locked fields when clicked and by three functions, `clickChunk`, `clickLine`, and `clickText`.

Here's an example in which you might use group text. You have a field with a list containing George Washington, King George, and George Bush, and you want to display more information about the appropriate George on the screen when his name is clicked. If these three phrases are set to plain text, clicking "George" wouldn't be specific enough, because the HyperTalk `clickText` function would only return the single word, George, without specifying more information about which George was clicked. If you set the style of each of the phrases George Bush, George Washington, and King George to `group`, then when the user clicks any word in the group phrase, the person's full name is returned and can be analyzed. If the user clicks either George or Bush in the phrase George Bush, the whole phrase—not just the word the user clicked—is returned by the `clickText` function.

For Paint text, you can also set the `textStyle` property from the Font dialog box, which is invoked by choosing Text Style from the Edit menu, by double-clicking the Paint Text tool, or by pressing Command-T while using a Paint tool.

For buttons or fields, you can also set the `textStyle` property by choosing a style from the Style menu or in the Font dialog box. To invoke the Font dialog box, you click the Font button in the Field or Button Info dialog box, choose Text Style from the Edit menu, or press Command-T while a button or field is selected.

If you reset the `textStyle` property for a field, any text that is already in that field is updated to the specified style.

If different styles of text are within a text selection, `textStyle` returns the result `mixed`.

See also the `printTextStyle` and `scriptTextStyle` properties, earlier in this chapter, and the `clickChunk`, `clickLine`, `clickText`, `selectedChunk`, and `selectedLine` functions in Chapter 11.

TextStyle (menu items)

Applies to Menu items

Syntax `set [the] textStyle of` *menuItem* `of` *menu* `to` *style*

MenuItem is an expression that yields a menu item descriptor. *Menu* is an expression that yields a menu descriptor. *Style* is an expression that yields a value of `plain` or any combination of the following: `bold`, `italic`, `underline`, `outline`, `shadow`, `condense`, and `extend` (separated by commas).

Examples

```
set the textStyle of menuItem "Get Back" of menu "Direction" ¬
to "outline"
put the textStyle of menuItem "Get Back" of menu "Direction"
```

Description You use the `textStyle` property to set or determine the text style of a specified menu item. The default value for the `textStyle` property is plain.

The `textStyle` property could be used with the `checkMark` property to indicate that a menu item has been chosen.

If you try to modify or determine the `textStyle` property of a menu item that does not exist, HyperCard displays a "No such menu item" dialog box.

Notes The text style of the menu items in the Font and Tools menu cannot be altered with the `textStyle` property.

See also the `printTextStyle` and `scriptTextStyle` properties, earlier in this chapter, and the `create menu` and `put` commands in Chapter 10, "Commands."

Top

Applies to Buttons, fields, windows

Syntax `set [the] top of` *object* `to` *number*

Object is an expression that yields one of the following objects in the current stack:

a valid button descriptor
a valid field descriptor
`tool window` or `window "tools"` (the Tools palette)
`pattern window` or `window "patterns"` (the Patterns palette)
`message [box]` or `message [window]`
`card window`
`scroll window` or `window "scroll"`
`window` *stackName*

Number is an expression that yields an integer that is the vertical offset in pixels of the top of the specified object. *StackName* is an expression that yields the name of an open stack.

Examples

```
set top of button 2 to 65
put top of button 2
set top of tool window to 100
```

Description You use the `top` property to determine or change the value of item 2 of the `rectangle` property (left, top, right, bottom) when applied to the specified object or window. The `top` property of an object can also be set to a value off the screen. Setting the `top` property of an object to a value off the screen will make the object seem as though it were hidden.

Note See also the `rectangle` property, earlier in this chapter.

TopLeft

Applies to Buttons, fields, windows

Syntax `set [the] topLeft of` *object* `to` *Point*

Object is an expression that yields one of the following objects in the current stack:

a valid button descriptor
a valid field descriptor
`tool window` or `window "tools"` (the Tools palette)
`pattern window` or `window "patterns"` (the Patterns palette)
`message [box]` or `message [window]`
`card window`
`scroll window` or `window "scroll"`
`window` *stackName*

Point is an expression that yields a list of two integers separated by a comma. *Point* represents the horizontal and vertical offsets, respectively, in pixels from the upper-left corner of the card to the upper-left corner of the specified object. *StackName* is an expression that yields the name of an open stack window.

Examples

```
set topLeft of bkgnd button id 23 to 64,30
put topLeft of window scroll
put the topLeft of Tool window
set topLeft of message box to 150,75
set the topLeft of window "Home" to "87,65"
```

Description You use the `topLeft` property to determine or change items 1 and 2 of the value of the `rectangle` property (left, top, right, bottom) when applied to the specified object or window.

Note See also the `rectangle` property, earlier in this chapter.

TraceDelay

Applies to Global environment

Syntax `set [the] traceDelay to` *number*

Number is an expresion that yields zero or a positive integer.

Example

```
set traceDelay to 32
put the traceDelay
```

Description You use the `traceDelay` property to set or retrieve the value of the built-in HyperCard debugger's trace rate. Setting the number value changes the number of ticks HyperCard waits between executing lines of HyperTalk while tracing. The default value for `traceDelay` is `0`, the fastest trace rate.

UserLevel

Applies to Global environment

Syntax `set userLevel to` *number*

Number is an expresion that yields one of the valid user-level numbers, `1` through `5`.

Example

```
set userLevel to 5
```

Description You use the `userLevel` property to set or retrieve the value of the current HyperCard user level. User levels give progressively more power to the user. The levels are 1 (Browsing), 2 (Typing), 3 (Painting), 4 (Authoring), and 5 (Scripting), as explained in the *HyperCard Reference*.

If you set the value of `userLevel` to a number lower than 1 or higher than 5, it automatically reverts to 1 or 5, respectively.

Notes You can invoke the Protect Stack dialog box from the File menu to impose a limit on the user level available in a stack. In that case, setting the user level higher than the Protect Stack limit has no effect, although it generates no error message. On leaving the protected stack, the user level in effect when the stack was entered is restored.

If your stack script changes the value of `userlevel`, be sure to restore the original value of `userlevel` when your stack closes.

UserModify

Applies to Global environment

Syntax `set userModify to` *boolean*

Boolean is an expression that yields either `true` or `false`.

Examples

```
set userModify to true
set userModify to false
```

Description The `userModify` property is a global property pertaining to HyperCard itself. It controls whether or not a user can type into fields or use Paint tools on a stack that has been write-protected. A stack is write-protected under the following circumstances:

- The stack is on a CD-ROM.
- The stack is on a file server in a folder whose access privileges are set to Read Only.
- The Locked box is checked in the stack's Get Info dialog box in the Finder File menu.
- The stack is on a locked 3.5-inch disk.
- "Can't modify stack" is checked in the stack's Protect Stack dialog box.

Script The following field script allows a user to type text into any field of a locked stack that contains the word "canwrite" in its script.

```
on openField
  set userModify to (script of the target contains "canwrite")
end openField
```

Note Changes can be made only to the level that the `userlevel` settings allow. Any changes made to the stack are disregarded when the stack is closed.

See also the `cantDelete` and `cantModify` properties, earlier in this chapter.

VariableWatcher

Applies to Global environment

Syntax

```
set [the] variableWatcher to name
```

Name is an expression that yields a valid variable watcher XCMD name.

Examples

```
set variableWatcher to "MyWatcher"
put the variableWatcher -- puts the name of the current
                        -- variable watcher in msg box
```

Description You use the `variableWatcher` property to determine or to change the current variable watcher. The default value for `variableWatcher` is `variableWatcher`, the built-in variable watcher. You display the current variable watcher with the `show` command or by setting the `visible` property of the variable watcher window to `true`.

The built-in variable watcher is a HyperCard XCMD. It can be replaced with a custom Variable Watcher XCMD by setting the `messageWatcher` property to the name of a custom variable watcher XCMD.

Note See also the description of the Variable Watcher in Chapter 3, "The Scripting Environment."

For more information about creating and calling a custom variable watcher XCMD, see Appendix A, "External Commands and Functions."

VBarLoc

Applies to Variable watcher windows

Syntax

```
set [the] vBarLoc of window "variable watcher" to number
```

Number is an expression that yields a positive integer that represents the offset in pixels from the left side of the variable watcher window to the vertical bar in the window.

Examples

```
set the vBarLoc of window "variable watcher" to 123

put the vBarLoc of window "variable watcher"
-- puts the current location of vertical bar in msg box
```

Description You use the `vBarLoc` property to determine or to change the current position of the vertical bar in the variable watcher window. The vertical bar separates the variable names from the actual values of the variables.

Notes The built-in variable watcher is a HyperCard XCMD. It can be replaced with a custom variable watcher XCMD by setting the `messageWatcher` property to the name of a variable watcher XCMD.

A custom variable watcher may or may not respond to the `vBarLoc` property. It is up to the variable watcher XCMD to provide support for variable watcher properties.

See also the description of the variable watcher in Chapter 3, "The Scripting Environment," and the `hBarLoc`, `rect`, and `variableWatcher` properties in this chapter.

For more information about creating and calling a custom variable watcher XCMD, see Appendix A, "External Commands and Functions."

Version

Applies to HyperCard, stacks

Syntax

```
the [long] version [of HyperCard]
the version of stack stackName
```

StackName is an expression that yields a stack name.

Example

```
if the version>1.0 then set textArrows to true
```

Description The `version` property returns the version number of the HyperCard application currently running or the versions of HyperCard that created and modified a specified stack.

The `long version` returns an 8-digit number that represents the major revision number, minor revision number, and software state (development, alpha, beta, or final, plus the release number). Here are the values the numbers represent:

version *xxyyzzrr*

xx major revision number
yy minor revision number
zz 80 = final
60 = beta
40 = alpha
20 = development
rr release number

For example, 0200600E is version 2.0 beta engineering release and 02008000 is version 2.0 final.

The `version of` *stackName* form returns a list of five comma-separated 8-digit numbers. The first four of these numbers are of the form described previously for the `long version`. They are respectively; the version of HyperCard used to create this stack; the version of HyperCard that last compacted this stack; the version of HyperCard that last modified the stack; and the version of HyperCard that first modified the stack. The last number is the date and time (in seconds) of the most recent save before the start of the current session. (You can use the `convert` command to change the seconds format into a date and time format.)

Visible

Applies to Buttons, fields, windows

Syntax `set the visible of` *object* `to` *boolean*

Object is an expression that yields one of the following objects in the current stack:

a valid button descriptor
a valid field descriptor
`message [box]` or `message [window]`
`pattern window` or `window "patterns"` (the Patterns palette)
`tool window` or `window "tools"` (the Tools palette)
`message watcher` or `window "message watcher"`
`variable watcher` or `window "variable watcher"`
`card window`
`scroll window` or `window "scroll"`
`window` *stackName*

Boolean is an expression that yields either `true` or `false`. *StackName* is an expression the yields the name of an open stack window.

Examples

```
set the visible of tool window to false
set the visible of window "variable watcher" to true
set the visible of msg to false
the visible of window "SnowFlakes"
set visible of button "now it's gone" to false
```

Description The `visible` property determines whether a button, field, or window is shown or hidden on the screen.

The Tools and Patterns palettes become visible when you tear them off the menu bar; the Message box can be toggled between being visible and hidden by pressing Command-M. The `visible` property applies only to buttons and fields on the current card.

Script

This script example sets the `visible` property of a background button in an `openCard` handler. This script would be useful in a situation where you have been using a group of background buttons to help the user navigate through a stack. For example, the buttons might be a go-forward button, a go-back button, and a return-to-start button. When the user reaches the last card in the stack, you should remove the go-forward button and leave only the go-back and return-to-start buttons as choices. This simple script would work in the last card or background.

```
on openCard
   if the name of this card is "Last one" then
      set visible of bkgnd button "go forward" to false else
      set visible of bkgnd button "go forward" to true
   end if
end openCard
```

The script that follows could be used to show a hidden field that is used for making notes. Create a button and a background field with the `sharedText` property set to `false`. The script placed in the button would display a field named Notes with the `show` command based on the value of the `visible` property of the Notes field. It also hides the field, if you click the button again.

```
on mouseUp -- button or field script to show a field
   if the visible of bkgnd field "Notes" is false then
      show bkgnd field "Notes" else
      hide bkgnd field "Notes"
   end if
end mouseUp
```

The next short script makes a field disappear after it has been made visible. The script, when placed in a locked field, sets the `visible` property of the field to `false` when the field receives a `mouseUp` message. Whenever a user clicks on the visible field it disappears.

```
on mouseUp -- field script to set visible property
   set the visible of me to not the visible of me
end mouseUp
```

Notes

Setting a window's `visible` property to `false` is the same as clicking its close box or hiding it with the `hide` command. You can read from and write to hidden fields. You cannot however, select the text of hidden fields. You can change properties of hidden fields just as you can visible fields.

You can't get the `visible` property of the Message Watcher or Variable Watcher windows, you can only set it.

See also the description of the Message Watcher and Variable Watcher in Chapter 3, "The Scripting Environment."

See also the `show` and `hide` commands in Chapter 10.

WideMargins

Applies to

Fields

Syntax

```
set [the] wideMargins of field to boolean
```

Field is an expression that yields a background or card field descriptor. *Boolean* is an expression that yields either `true` or `false`.

Example

```
set wideMargins of field "just fine" to true
the wideMargins of field 1 -- puts current value in msg
```

Description

You use the `wideMargins` property to specify whether some extra space is included at the left and right side of each line in the field (to make the text easier to read). The default value of `wideMargins` is `false`.

Note

You can also change this property by clicking the Wide Margins check box in the Field Info dialog box.

Width

Applies to Buttons, fields

Syntax

```
set [the] width of object to number
```

Object is an expression that yields a valid button or field descriptor. *Number* is an expression that yields a pGositive integer. The *number* value represents the total number of pixels in the horizontal width of the specified object.

Examples

```
set width of button id 128 to 104
put width of button 4 -- puts width in msg
the width of bkgnd field "phoneList" -- puts width in msg
```

Description You use the `width` property to determine or change the horizontal distance in pixels occupied by the rectangle of the specified button or field. You can get or put the width of windows, but you can't set the width of any window including the Message box.

When you set the width of a button or field, its `location` property (center coordinate) remains the same.

Note See also the `rectangle` property, earlier in this chapter.

Appendix A External Commands and Functions

This appendix describes the external command and function interface of HyperCard. In addition to general information about external commands and functions, this appendix contains specific information that requires a reading knowledge of 68000 assembly language, Pascal, or C to be understood. This appendix does not include information about how to write code, nor does it explain how to use a compiler or assembler to create an executable resource.

Definitions, uses, and examples

External commands and functions are extensions to the HyperTalk built-in command and function set. HyperCard includes interface procedures that make extending HyperTalk in this way convenient and practical for expert programmers.

XCMD and XFCN resources

External commands, *ex-commands* (XCMDs) and *ex-functions* (XFCNs), are executable Macintosh code resources, written in a Macintosh programming language (such as Pascal, C, or 68000 assembly language), which are attached to the HyperCard application or a stack with a resource editor such as ResEdit. The resource type of an external command is 'XCMD' and the resource type of an external function is 'XFCN'.

A Macintosh XCMD or XFCN is a compiled (or assembled) executable code module. After XCMDs or XFCNs have been created and attached to HyperCard or a stack, they're called from HyperTalk in much the same way that built-in commands or user-defined message and function handlers are called. They also use the message-passing hierarchy in the same way, and they communicate with HyperCard through the same parameter block data structure.

An XCMD or XFCN resource has no header bytes; it is invoked by a jump instruction to its entry point. These resources are simpler than Macintosh drivers: they can't have any global (or static) data, and they can't be larger than 32 KB in size. (For more details about these restrictions, see "Guidelines for Writing XCMDs and XFCNs," later in this appendix.)

For detailed information on Macintosh resources, see *Inside Macintosh*, published by Addison-Wesley.

Uses for XCMDs and XFCNs

External commands and functions can provide access to the Macintosh Toolbox and to some of the HyperCard internal routines; they can provide fast processing speed for time-critical operations; and they can override built-in HyperTalk commands to provide custom solutions. XCMDs or XFCNs can be used for serial port input and output routines, custom search-and-replace routines, AppleCD SC control routines, file input and output routines, and so on.

A typical use for an XCMD would be as an interface for a driver, allowing HyperCard to control an external device such as a video disc player. Such an interface would have three parts: the driver, the XCMD, and a HyperTalk handler. The driver would be completely separate from HyperCard. (See Volume II of *Inside Macintosh* for information about writing drivers.) The XCMD would be small; its purpose would be to convert HyperTalk messages to the appropriate driver calls. The HyperTalk handler would call the XCMD with various parameters directing it to open or close the driver or to perform a specific control call.

Using an XCMD or XFCN

You invoke XCMDs and XFCNs from HyperTalk using the regular message syntax and user-defined function call syntax. The message or function call is passed through the HyperCard message-passing hierarchy.

Invoking XCMDs and XFCNs

You invoke an XCMD as you do a message handler. That is, you type the name of the XCMD followed by its parameters in a HyperTalk script or in the Message box. Separate the parameters (if more than one) with commas, and put quotation marks around parameters of more than one word. When the script executes or when you send the Message box contents by pressing Return or Enter, HyperCard sends the message through the normal message-passing hierarchy. For external commands, the Macintosh resource name correlates to the message name—the first word in the message.

Similarly, you call an XFCN in a HyperTalk statement in the same way you would a user-defined function. (Use parentheses after the function, rather than preceding the function with word `the`.) Enclose any parameters within parentheses, separate them (if more than one) with commas, and put quotation marks around parameters of more than one word. If the function takes no parameters, append empty parentheses after it. For external functions, the Macintosh resource name correlates to the function name—the word preceding parentheses in the function call.

You can pass a maximum of 16 parameters to an XCMD or XFCN.

Message-passing hierarchy

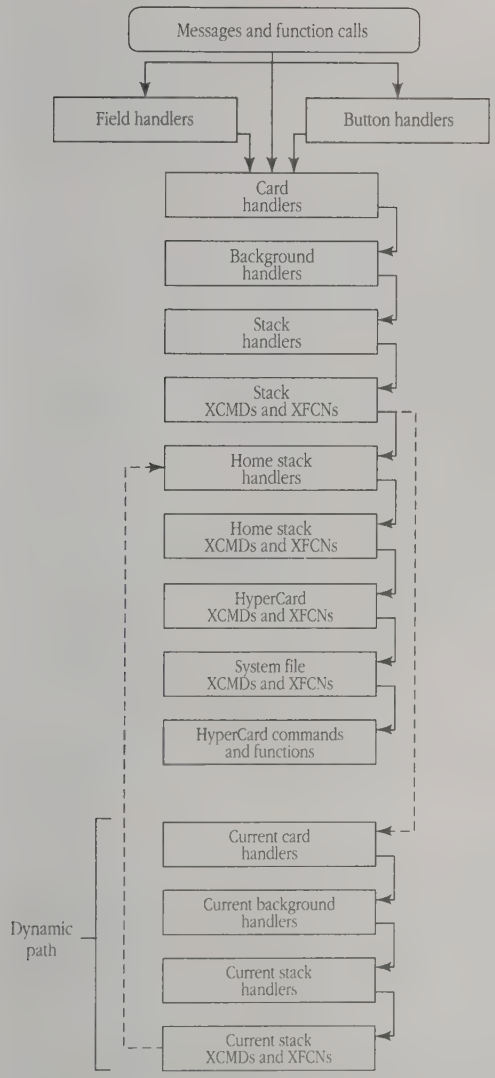
External commands and functions use the message-passing hierarchy in the same way as message and function handlers and built-in commands and functions. External commands and functions can be attached to any stack or to the HyperCard application.

If a stack receives a message or function call for which it has no handler, then before passing the message or function call to another stack (if added to the message-passing hierarchy with the `start using` command) or HyperCard, it checks to see if it has an external command or function of the same name. When HyperCard receives a message or function call, it checks to see if it has an external command or function *before* it looks for a built-in command or function.

That is, HyperCard searches for message and function handlers, XCMDs and XFCNs, and built-in commands and functions through the message-passing hierarchy shown in Figure A-1.

Chapter 4 discusses the message-passing hierarchy, including the dynamic path, in detail.

■ **Figure A-1** Message-passing hierarchy, including XCMDs and XFCNs



Guidelines for writing XCMDs and XFCNs

XCMDs and XFCNs can call most of the Macintosh Toolbox traps and routines, but they have certain limitations and restrictions. They can't do everything that an application can do because they are guests in HyperCard memory space. In that regard they are more like desk accessories than applications. Here are some guidelines for writing XCMDs and XFCNs:

- Do not initialize the various Macintosh managers by calling their initialization routines. That is, don't call `InitGraf`, `InitFonts`, `InitWindows`, and so on.
- Do not rely upon having lots of RAM available for your XCMD. There is some extra space in the HyperCard heap, but if HyperCard is running in 750 KB under MultiFinder, for example, an XCMD should not be bigger than about 32 KB.
- Do not use register A5 of a 68000-family processor. The value in A5 belongs to HyperCard, and it points to HyperCard global data, jump table, and other things that constitute an "A5 world." XCMDs do not have their own A5 world.
- XCMDs cannot have global data.
- You can use string literals in XCMDs compiled with the `-b` switch in MPW C version 3.0 or later. You can use 'STR' resources or put the strings in a short assembly-language glue file.
- XCMDs cannot have a jump table, so they cannot have code segments. This restriction imposes a 32 KB limit on the size of XCMDs for 68000-based machines. (The 68020 supports longer branches.)
- XCMDs can, however, allocate small chunks of memory by standard `NewHandle` calls. (You can also allocate memory with `NewPtr` calls, but they should be used sparingly to avoid heap fragmentation.)
- If your XCMD allocates some memory in the heap, it should also deallocate the memory.
- If an XCMD allocates a handle to save state information between invocations of the XCMD, then you can pass the handle back to HyperCard to be stored somewhere in the current stack, such as in a hidden field or in a HyperTalk global variable. You must convert the handle to a hex string, because all values are treated as strings by HyperTalk.
- Since HyperCard jumps to the start of an XCMD's code, the XCMD's entry point must be the start of the XCMD. In other words, the XCMD glue must follow the main routine, so the link order is vitally important.
- If the size of your XCMD approaches 32 KB, consider converting it to a device driver.

Attaching an XCMD or XFCN

To attach an existing XCMD or XFCN (one that has already been compiled or assembled into a resource) to one of your stacks, use a resource editor such as ResEdit or obtain one of the public domain stacks for moving resources, for example, the ResCopystack. The following steps describe the procedure using ResEdit:

1. Launch ResEdit.
2. Select and open the stack containing the 'XCMD' or 'XFCN' resource you want.
3. Select and open the resource type of 'XCMD' or 'XFCN'.
4. Select and open the particular resource you want by name.
5. Press Command-C to copy the resource.
6. Select and open the stack into which you want to paste the resource.
7. If your stack has no resource fork, ResEdit displays a dialog box asking if you want to open one. Click OK. ResEdit opens a window.
8. Press Command-V to paste the resource into your stack.
9. Click the window's close box. When ResEdit asks if you want to save the file, click Yes.
10. Quit ResEdit.

Parameter block data structure

If HyperCard matches a message or function call with an external command or function, it passes a single argument to the XCMD or XFCN: a pointer to a parameter block called `XCmdBlock`. All communication between HyperCard and the XCMD or XFCN passes through the parameter block. In Pascal, the parameter block data structure is a record; in C it's a struct.

HyperCard uses the first two fields of the parameter block to pass information to the XCMD or XFCN before invoking its execution. The XCMD or XFCN uses the other data fields in `XCmdBlock` to pass back results and to communicate with HyperCard during execution.

The Pascal parameter block is shown below:

```
TYPE

    XCmdPtr = ^XCmdBlock;
    XCmdBlock = RECORD
                paramCount:  INTEGER; { If -1 then this is an
                                        event handling call. }
                params:      ARRAY[1..16] OF Handle;
                returnValue: Handle;
                passFlag:    BOOLEAN;
                entryPoint:  ProcPtr; { to call back to HyperCard }
                request:     INTEGER;
                result:      INTEGER;
                inArgs:      ARRAY[1..8] OF LongInt;
                outArgs:     ARRAY[1..4] OF LongInt;
            END;

END;
```

Passing parameters to XCMDs and XFCNs

Before calling the XCMD or XFCN, HyperCard places the number of parameters and handles to the parameter strings in two fields of the parameter block: `paramCount` and `params`.

ParamCount

HyperCard puts an integer representing the parameter count in field `paramCount`. You can pass a maximum of 16 parameter strings.

Params

HyperCard evaluates the parameters and puts their values into memory as zero-terminated ASCII strings. Before it invokes the XCMD or XFCN, HyperCard puts the handles to the parameter strings into the `params` array. For example, the command `Beep 5` creates a single handle in `params[1]` containing ($35) the ASCII equivalent of `5` and a zero terminator. HyperCard disposes of the handle.

Passing back results to HyperCard

When an XCMD or XFCN finishes executing, HyperCard examines two fields of the parameter block: `returnValue` and `passFlag`.

ReturnValue

An XCMD or XFCN can store one zero-terminated string to communicate the result of its execution. HyperCard looks for a handle to the result string in the `returnValue` field of `XCmdBlock`. Storing a result string is optional for an XCMD; it is expected of an XFCN, but it's not required. If you store a result string handle in `returnValue` in an XCMD, the user can get it by using the HyperTalk function `the result` (useful for explaining why there was an error). For an XFCN, HyperCard uses the `returnValue` string to replace the function call itself in the HyperTalk statement containing the call. If you don't store anything, `the result` is the empty string.

PassFlag

When an XCMD or XFCN terminates, HyperCard examines the Boolean value of the `passFlag` field. If `passFlag` is false (the normal case), control passes back to the previously executing handler (or to the HyperCard idle state if no handler was executing). If `passFlag` is true, HyperCard passes the message or function call to the next object in the hierarchy. This has the same effect as the `pass` control statement in a script.

Callbacks

The remaining five fields of the `XCmdBlock` record have to do with calling HyperCard back in the middle of execution of an XCMD or XFCN. You use the callback mechanism to obtain data or request HyperCard to perform an action. HyperCard has 77 callback requests. (See "Callback Procedures and Functions," later in this appendix.) The five `XCmdBlock` fields that compose the callback interface are `entryPoint`, `request`, `result`, `inArgs`, and `outArgs`.

EntryPoint

When HyperCard sets up the parameter block data structure before passing control to an XCMD or XFCN, it places an address in `entryPoint`. The XCMD or XFCN uses this address to execute a jump instruction to pass control to HyperCard for the callback.

Request

Before executing the jump instruction, the XCMD or XFCN puts an integer representing the callback request it's making into the `request` field. The request codes are listed in "Callback Procedures and Functions," later in this appendix.

Result

After it completes the callback request, HyperCard places an integer result code in the `result` field. The result code can be 0, 1, or 2. If the callback executed successfully, the result is 0; if it failed, the result is 1; if the callback request is not implemented in HyperCard, the result is 2.

InArgs

The XCMD or XFCN sends up to eight arguments to HyperCard as long integers in the `inArgs` array. Depending on the callback request, HyperCard expects arguments in certain elements of the `inArgs` array. The callback arguments are shown in Pascal in "Callback Procedures and Functions," later in this appendix.

OutArgs

After it executes the callback request, HyperCard returns up to four long integers (or other types, such as handles) to the XCMD or XFCN as elements of the `outArgs` array. The arguments HyperCard returns from callbacks are shown in Pascal in "Callback Procedures and Functions," later in this appendix.

Callback procedures and functions

If you want to manage a callback to HyperCard yourself, you can define the `XCmdBlock` data structure in your XCMD or XFCN. Then you can put values you want to send to HyperCard in `inArgs`, put a request code in `request`, and execute a jump instruction to the address HyperCard places in `entryPoint`. HyperCard returns values in `outArgs` and a result code in `result`.

However, if you use MPW Pascal or C, you can take advantage of interface definition and glue files. The definition and glue files provide simple procedure and function calls that you can use inside your XCMD or XFCN to handle callback requests more easily. Include them when you compile your XCMD or XFCN.

The Pascal code for an XCMD or XFCN should include the definition file `HyperXCmd.p` at the beginning of the `USES` clause and the glue file `XCmdGlue.inc` at the end with the `$I` directive. There must be an argument of type `XCmdPtr` passed by HyperCard to the XCMD or XFCN. In the glue routines, all strings are Pascal strings unless noted as zero-terminated strings (which have no length byte, the end of the string being indicated by a null byte). In general, if a handle is returned, the XCMD or XFCN is responsible for disposing of it.

✍ The HyperCard callbacks new since the first edition of this book are denoted by the symbol at the beginning of this paragraph.

HyperTalk utilities

```
FUNCTION EvalExpr(paramPtr: XCmdPtr; expr: Str255): Handle;
```

`EvalExpr` evaluates the HyperTalk expression passed in `expr` and returns a handle to a zero-terminated string containing the result of the evaluation. For example, `EvalExpr ('the long date')` returns a handle to a string containing the current date in the long format (`Saturday, June 25, 1988`). The caller must dispose of the handle.

✍ `PROCEDURE RunHandler(paramPtr: XCmdPtr; handler: Handle);`

The zero-terminated string in `handler` is interpreted first as a message. If it is a message (command or function), it is sent to the current card. The text can be one or more lines of HyperTalk, including conditional statements and repeat loops. The lines are executed as though sent from the Message box. If it is multiple lines beginning with `on` *messageName* and ending with `end` *messageName*, *messageName* is sent to this handler in the context of the card. If the handler exits, execution terminates. If the handler contains the line `pass` *messageName*, the *messageName* is passed down the normal HyperTalk message-passing path, beginning with the current card script.

You cannot override a script executed with `RunHandler`. For example, if the current card has an `on` *messageName* handler, and an XCMD issues a `runHandler` callback with an `on` *messageName*, the XCMD's handler executes. If the handler passes the message, the card's script executes.

Execution is somewhat slower using `RunHandler` than it would be running the same script as a card handler. You cannot use the debugging tools to debug scripts executed with `runHandler`.

```
PROCEDURE SendCardMessage(paramPtr: XCmdPtr; msg: Str255);
```

The string in `msg` is sent as a message to the current card.

```
PROCEDURE SendHCMessage(paramPtr: XCmdPtr; msg: Str255);
```

The string in `msg` is sent as a message directly to HyperCard, bypassing the entire message-passing hierarchy.

Memory utilities

```
FUNCTION GetGlobal(paramPtr: XCmdPtr; globName: Str255): Handle;
```

`GetGlobal` returns a handle to a zero-terminated string that contains a copy of the contents of the HyperTalk global variable `globName`. If `globName` doesn't exist, `GetGlobal` returns a handle to an empty string. The caller must dispose of the handle.

```
PROCEDURE SetGlobal(paramPtr: XCmdPtr; globName: Str255; globValue:
                    Handle);
```

`SetGlobal` copies the zero-terminated string to which `globValue` is a handle into the HyperTalk global variable named `globName`. If `globName` doesn't exist, `SetGlobal` creates it. HyperCard does not dispose of `globValue`.

```
PROCEDURE ZeroBytes(paramPtr: XCmdPtr; dstPtr: Ptr; longCount: LongInt);
```

`ZeroBytes` sets `longCount` bytes beginning at `dstPtr` to 0. It performs no boundary checking. For example, it can write past the end of a zero-terminated string.

String utilities

```
PROCEDURE ScanToReturn(paramPtr: XCmdPtr; VAR scanPtr: Ptr);
```

`ScanToReturn` scans the zero-terminated string pointed to by `scanPtr`, stopping just past the first return character or at the end of the string. `ScanPtr` is incremented to point to the new location.

```
PROCEDURE ScanToZero(paramPtr: XCmdPtr; VAR scanPtr: Ptr);
```

`ScanToZero` scans the zero-terminated string pointed to by `scanPtr`, stopping at the end of the string. `ScanPtr` is incremented to point to the new location.

```
FUNCTION StringEqual(paramPtr: XCmdPtr; str1,str2: Str255): BOOLEAN;
```

`StringEqual` compares the two Pascal strings `str1` and `str2` (case-insensitive and diacritical-sensitive) and returns `TRUE` if the two strings are identical; otherwise it returns `FALSE`.

```
FUNCTION StringLength(paramPtr: XCmdPtr; strPtr: Ptr): LongInt;
```

`StringLength` returns the number of characters in the zero-terminated string pointed to by `strPtr`. Note that `strPtr` is a pointer, not a handle.

```
FUNCTION StringMatch(paramPtr: XCmdPtr; pattern: Str255; target: Ptr):
                    Ptr;
```

`StringMatch` performs a case-insensitive search for `pattern` (a Pascal string) in the zero-terminated string pointed to by `target`. If the search is successful, the location of the first matching character is returned as the function result. If the search is unsuccessful, `StringMatch` returns `nil`. This is equivalent to the HyperTalk `offset` function.

```
PROCEDURE ZeroTermHandle(paramPtr: XCmdPtr; hndl: Handle);
```

`ZeroTermHandle` expands the block referred to `hndl` by one byte and then sets the extra byte to 0, making `hndl` legal for operations such as `SetGlobal`, `FormatScript`, and `ZeroToPas`.

String conversions

```
PROCEDURE BoolToStr(paramPtr: XCmdPtr; bool: BOOLEAN; VAR str: Str255);
```

`BoolToStr` converts `bool` to a Pascal string (`"true"` or `"false"`).

```
PROCEDURE ExtToStr(paramPtr: XCmdPtr; num: Extended; VAR str: Str255);
```

`ExtToStr` converts `num` (a SANE extended type) to a Pascal string.

```
PROCEDURE LongToStr(paramPtr: XCmdPtr; posNum: LongInt; VAR str:
                    Str255);
```

`LongToStr` converts `posNum` (a 32-bit unsigned integer) to a Pascal string.

```
PROCEDURE NumToHex(paramPtr: XCmdPtr; num: LongInt; nDigits: INTEGER;
                   VAR str: Str255);
```

`NumToHex` returns in `str` a hexadecimal (base 16) representation of the value of `num`, expanding the string to `nDigits` in length.

```
PROCEDURE NumToStr(paramPtr: XCmdPtr; num: LongInt; VAR str: Str255);
```

`NumToStr` converts `num` (a 32-bit signed integer) to a Pascal string.

```
FUNCTION PasToZero(paramPtr: XCmdPtr; str: Str255): Handle;
```

`PasToZero` converts `str` to a zero-terminated string and returns a handle to the new string. The caller must dispose of the handle.

```
PROCEDURE PointToStr(paramPtr: XCmdPtr; pt: Point; VAR str: Str255);
```

`PointToStr` converts the point passed in `pt` to a Pascal string and returns the string in `str`.

```
PROCEDURE RectToStr(paramPtr: XCmdPtr; rct: Rect; VAR str: Str255);
```

`RectToStr` converts the rectangle passed in `rct` to a Pascal string and returns the string in `str`.

```
PROCEDURE ReturnToPas(paramPtr: XCmdPtr; zeroStr: Ptr; VAR str: Str255);
```

`ReturnToPas` copies characters from the zero-terminated string pointed to by `zeroStr` into the Pascal string `pasStr`, stopping at the first return character (`ASCII $0D`), the end of the zero-terminated string, or the 255th character, whichever comes first.

```
FUNCTION StrToBool(paramPtr: XCmdPtr; str: Str31): BOOLEAN;
```

`StrToBool` converts `str` to a Boolean (`TRUE` or `FALSE`) and returns the Boolean value as its result.

```
FUNCTION StrToExt(paramPtr: XCmdPtr; str: Str31): Extended;
```

`StrToExt` converts `str` to an extended type. Extended numbers contain a sign bit, 15 bits for the exponent, and 63 bits for the significant. This is the standard data type for SANE, the Standard Apple Numeric Environment.

```
FUNCTION StrToLong(paramPtr: XCmdPtr; str: Str31): LongInt;
```

`StrToLong` converts `str` to a long (32-bit) unsigned integer. Unsigned long integers range from
0 to 4,294,836,225.

```
FUNCTION StrToNum(paramPtr: XCmdPtr; str: Str31): LongInt;
```

`StrToNum` converts `str` to a long (32-bit) signed integer. Signed long integers range from (-2,147,418,112) to 2,147,418,112.

```
PROCEDURE StrToPoint(paramPtr: XCmdPtr; str: Str255; VAR pt: Point);
```

`StrToPoint` converts the Pascal string passed in `str` to a point and returns the point in `pt`.

```
PROCEDURE StrToRect(paramPtr: XCmdPtr; str: Str255; VAR rct: Rect);
```

`StrToRect` converts the Pascal string passed in `str` to a rectangle and returns the rectangle in `rct`.

```
PROCEDURE ZeroToPas(paramPtr: XCmdPtr; zeroStr: Ptr; VAR pasStr: Str255);
```

`ZeroToPas` converts the zero-terminated string pointed to by `zeroStr` to a Pascal string and returns the string in `pasStr`. The caller must dispose of the handle.

Field utilities

```
FUNCTION GetFieldByID(paramPtr: XCmdPtr; cardFld: BOOLEAN; fldID:
                      INTEGER): Handle;
```

`GetFieldByID` returns a handle to a zero-terminated string that contains a copy of the contents of field ID `fldID`. If `cardFld` is `TRUE`, `fldID` is a card field; otherwise it is a background field.

```
FUNCTION GetFieldByName(paramPtr: XCmdPtr; cardFld: BOOLEAN; fldName:
                        Str255): Handle;
```

`GetFieldByName` returns a handle to a zero-terminated string that contains a copy of the contents of field `fldName`. If `cardFld` is `TRUE`, `fldName` is a card field; otherwise it is a background field.

```
FUNCTION GetFieldByNum(paramPtr: XCmdPtr; cardFld: BOOLEAN; fldNum:
                       INTEGER): Handle;
```

`GetFieldByNum` returns a handle to a zero-terminated string that contains a copy of the contents of field number `fldNum`. If `cardFld` is `TRUE`, `fldNum` is a card field; otherwise it is a background field.

```
FUNCTION GetFieldTE(paramPtr: XCmdPtr; cardFieldFlag: BOOLEAN;
                    fieldID, fieldNum:INTEGER; fieldNamePtr:
                    StringPtr): TEHandle;
```

To determine which field the XCMD wants to manipulate, HyperTalk uses the following logic: if `fieldID` is non-zero, then HyperTalk uses it; else if `fieldNum` is non-zero, then HyperTalk uses it; else if `fieldNamePtr` is not `NIL`, HyperTalk uses the field name pointed to by it. `GetFieldTE` returns a copy of the styled `TEHandle` from the specified field, including style runs (see *Inside Macintosh*, Volume V). The caller must dispose of this TEHandle. If `GetFieldTE` returns `NIL`, the field was not found or there wasn't enough memory to copy the text and styles. The XCMD should not use the `TEHandle` returned by this callback before adjusting the `TEHandle^^.inPort` field to point to another grafPort. HyperTalk sets this

field to point to the card window when this callback returns, so routines like `TEUpdate` and `TESetSelect` may cause the text to draw into the card window.

```
PROCEDURE SetFieldByID(paramPtr: XCmdPtr; cardFld: BOOLEAN; fldID:
                        INTEGER; fldVal: Handle);
```

`SetFieldByID` copies the zero-terminated string to which `fldVal` is a handle into the field ID `fldID`. If `cardFld` is `TRUE`, `fldID` is a card field; otherwise it is a background field. The caller must dispose of the handle.

```
PROCEDURE SetFieldByName(paramPtr: XCmdPtr; cardFld: BOOLEAN; fldName:
                         Str255; fldVal: Handle);
```

`SetFieldByName` copies the zero-terminated string to which `fldVal` is a handle into field `fldName`. If `cardFld` is `TRUE`, `fldName` is a card field; otherwise it is a background field. The caller must dispose of the handle.

```
PROCEDURE SetFieldByNum(paramPtr: XCmdPtr; cardFld: BOOLEAN; fldNum:
                        INTEGER; fldVal: Handle);
```

`SetFieldByNum` copies the zero-terminated string to which `fldVal` is a handle into the field number `fldNum`. If `cardFld` is `TRUE`, `fldNum` is a card field; otherwise it is a background field. The caller must dispose of the handle.

```
PROCEDURE SetFieldTE(paramPtr: XCmdPtr; cardFieldFlag: BOOLEAN;
                     fieldID, fieldNum: INTEGER; fieldNamePtr:
                     StringPtr; fieldTE: TEHandle);
```

`SetFieldTE` sets the text and styles of the field to the text and styles contained in `fieldTE`. The caller must dispose of this TEHandle.

```
PROCEDURE BeginXSound(paramPtr: XCmdPtr; window: WindowPtr);
```

`BeginXSound` informs HyperCard that an XCMD is about to use the Sound Manager. An XCMD should call `BeginXSound` before it attempts to allocate a sound channel or perform any other Sound Manager operation. After an XCMD calls `BeginXSound`, the HyperCard built-in `play` command will not operate until the XCMD calls `EndXSound` (see the next procedure). If an external window is making the callback, it should pass a pointer to its window in the `window` parameter. An XCMD should pass `NIL` for the `window` parameter. An XCMD or an external window can optionally pass a pointer to a valid external window in the `window` parameter if it wants to direct the call to another external window. If HyperTalk gets a valid `windowPtr` in `window`, it will post an `xGiveUpSoundEvt` to that window at an appropriate time. If the XCMD passes NIL for `window`, HyperCard will be unable to signal the XCMD when HyperCard needs the sound channel back.

An XCMD that uses the Sound Manager can be structured as follows:

```
BeginXSound(paramPtr);
(* allocate a sound channel *)
(* do your sound thing *)
(* deallocate sound channel *)
EndXSound(paramPtr);
```

```
PROCEDURE EndXSound(paramPtr: XCmdPtr);
```

`EndXSound` informs HyperCard that an XCMD has finished using the Sound Manager. If the XCMD has not previously called `BeginXSound`, `EndXSound` does nothing.

```
FUNCTION GetFilePath(paramPtr: XCmdPtr; fileName: Str255; numTypes: INTEGER;
                     typeList: SFTypeList; askUser: BOOLEAN;
                     VAR fileType: OSType; VAR fullName: Str255): BOOLEAN;
```

GetFilePath determines the full pathname of the file fileName using the search paths stored in the Home stack. If fileType is 'STAK', GetFilePath uses the stack search paths. If fileType is 'APPL', GetFilePath uses the application search paths. If fileType is neither 'STAK' nor 'APPL', GetFilePath uses the document search paths. If matchAnyType is TRUE, fileType is ignored in the search; otherwise the search is restricted to files of the type specified in fileType. NumTypes and typeList are used as they are described in *Inside Macintosh*, Volume I, page 523. If askUser is TRUE and HyperCard fails to find the file on its own, GetFilePath prompts the user to find the file with a directory dialog box. If the file is located either by HyperCard or by the user, the full pathname of the file is returned in fullName and the file type is returned in fileType. If the user clicks Cancel in the directory dialog box or HyperCard fails to find the file for any other reason, GetFilePath returns FALSE.

```
PROCEDURE GetXResInfo(paramPtr: XCmdPtr; VAR resFile: INTEGER;
                      VAR resID: INTEGER; VAR rType: ResType;
                      VAR name: Str255);
```

In resFile, GetXResInfo returns the file reference number of the resource file from which the calling XCMD was read in resFile, the resource ID of the XCMD in resID, the resource type (XCMD or XFCN) in rType, and the resource name in name.

```
PROCEDURE Notify(paramPtr: XCmdPtr);
```

If HyperCard is active or MultiFinder is not loaded, Notify returns immediately. Otherwise, Notify blinks the small HyperCard icon over the Apple in the Apple menu until the user opens the Apple menu and switches to HyperCard. Only then does Notify return. No other HyperCard processing takes place while Notify is waiting.

```
PROCEDURE SendHCEvent(paramPtr: XCmdPtr; event: EventRecord);
```

SendHCEvent is useful only to XCMDs that call the ToolBox routines GetNextEvent or WaitNextEvent. Such XCMDs should use SendHCEvent to pass events required by HyperCard. For example, an XCMD that creates a draggable window and calls GetNextEvent may receive update events for HyperCard windows. HyperCard performs the updates if it receives the update events via this callback. More importantly, an XCMD that calls GetNextEvent and receives an app4Evt generated by MultiFinder must pass the event along to HyperCard. *If HyperCard does not receive its suspend and resume events, unexpected results may occur.* XCMDs that create windows by means of the NewXWindow callback don't need to call GetNextEvent or WaitNextEvent and therefore don't need to use SendHCEvent.

```
PROCEDURE SendWindowMessage(paramPtr: XCmdPtr; windPtr: WindowPtr;
                            windowName: Str255; msg: Str255);
```

`SendWindowMessage` is functionally equivalent to `send` *message* `to window` *windowName* from HyperTalk. Use this for direct communication between XCMDs that manage external windows. If `windPtr` is not `NIL`, the window pointer in `windPtr` is used to determine which window receives the message, otherwise the name in `windowName` is used.

```
FUNCTION FrontDocWindow(paramPtr: XCmdPtr): WindowPtr;
```

`FrontDocWindow` returns the `WindowPtr` of the document that is frontmost in the HyperCard document layer. It does not return the `WindowPtr` of a window in the floating layer. To return the frontmost window of any type, use the Window Manager's `FrontWindow` function.

```
FUNCTION StackNameToNum(paramPtr: XCmdPtr; stackName: Str255): LongInt;
```

Internally, HyperCard no longer remembers stacks only by their name. It uses a stack number to represent the stack. This number is similar to a volume reference number; it is valid as an indicator as long as the application is open, but won't be valid across multiple launches. This number is valid when used in an `XTalkObject` to get and set the scripts of objects. `StackNameToNum` translates the name of a stack into this number.

```
FUNCTION ShowHCAlert(paramPtr: XCmdPtr; dlgID: INTEGER; promptStr: Str255): INTEGE
```

HyperCard contains four standard alert windows. XCMDs can use these, along with all of the smart window-centering, text-expanding, and command-key handling without having to use the answer and ask commands any more. `DlgID` must be one of the following constants:

```
errorDlgID          = 1;    { 1:OK (default) }
confirmDlgID        = 2;    { 1:OK (default) and 2:Cancel }
confirmDelDlgID     = 3;    { 1:Cancel (default) and 2:Delete }
yesNoCancelDlgID    = 4;    { 1:Yes (default), 2:Cancel, and 3:No }
```

The values listed in the comments correspond to the function results returned by `ShowHCAlert`. For example,

```
CASE ShowHCAlert(paramPtr,yesNoCancelDlgID,'Delete file "Fred?"') OF
  1: DeleteTheFile;       { user chose "OK" }
  2: Abort;               { user chose "Cancel" }
  3: Continue;            { user chose "No" }
END;
```

The pascal string in `promptStr` can contain the Dialog Manager's wildcard characters (`^0`, `^1`, `^2`, and `^3`) if the XCMD calls `ParamText` before calling `ShowHCAlert`.

Creating and disposing of external windows

```
FUNCTION GetNewXWindow(paramPtr: XCmdPtr; templateType: ResType;
                       templateID: INTEGER; colorWind: BOOLEAN;
                       floating: BOOLEAN): WindowPtr;
```

`GetNewXWindow` creates a new window or dialog box from a resource. If the Window Manager fails to create the window, `GetNewXWindow` returns `NIL`. If the window is created successfully, `GetNewXWindow` sets up the mechanism by which events pertaining to the window are sent to the XCMD that created it.

`TemplateType` must be either `'WIND'` or `'DLOG'`. `TemplateID` is the resource ID of the window or dialog box template to be used. The template resource can exist in any resource file that's currently open.

If `colorWind` is `TRUE`, HyperTalk attempts to create a color window using the `GetNewCWindow` or `NewCDialog` Toolbox trap. If `colorWind` is `TRUE` and Color QuickDraw is not present, the window is not created and `GetNewXWindow` returns `NIL`.

If `floating` is `TRUE`, HyperTalk places the new window in the floating layer. Otherwise, the window is placed in the document layer. See "Window Layer Management," later in this appendix, for an explanation of these two layers and their relationship to each other.

- *Note:* GetNewXWindow is compatible with nonstandard window definition functions (see NewXWindow for more information).

```
FUNCTION NewXWindow(paramPtr: XCmdPtr; boundsRect: Rect; title:
                    Str255; visible: BOOLEAN; procID: INTEGER;
                    colorWind: BOOLEAN; floating: BOOLEAN):
                    WindowPtr;
```

`NewXWindow` creates a new window and returns a pointer to it as the function's result. If the Window Manager fails to create the window, `NewXWindow` returns `NIL`. If the window is created successfully, `NewXWindow` sets up the mechanism by which events pertaining to the window are sent to the XCMD that created it.

If `colorWind` is `TRUE`, HyperTalk attempts to create a color window using the `NewCWindow` Toolbox trap. If `colorWind` is `TRUE` and Color QuickDraw is not present, the window is not created and `NewXWindow` returns `NIL`.

The value in `procID` is the same as the `procID` argument to the Window Manager routine `NewWindow`. For example, passing `documentProc` as the `procID` produces a standard document window with no zoom box. `BoundsRect` is the bounding rectangle for the window in global coordinates. `Title` becomes the title of the window.

The `visible` argument determines whether the window is visible when created.

To create windows similar to HyperCard palettes, such as the Tools palette and the Message box, use the HyperCard built-in window definition function for windows. Here are the possible built-in window values to use for `procID`:

```
paletteProc         = 2048;      { Window with grow box }
palNoGrowProc       = 2052;      { Standard Window }
palZoomProc         = 2056;      { Window with zoom and grow }
palZoomNoGrow       = 2060;      { Window with zoom and no grow }
hasZoom             = 8;
hasTallTBar         = 2;
toggleHilite        = 1;
```

For example, the navigator palette uses `palNoGrowProc` for `procID`.

- *Note:* `NewXWindow` is also compatible with other nonstandard window definition functions.

```
PROCEDURE CloseXWindow(paramPtr: XCmdPtr; window: WindowPtr);
```

When an XCMD that manages an external window requests that the window be closed, it should call `CloseXWindow`. When all pending calls to the XCMD have returned, HyperCard sends an `xCloseEvt` to the XCMD. Only in response to this event should it dispose of its data structures and exit.

When the external window receives the `xCloseEvt`, it signals its willingness to close by setting `paramPtr^.passFlag` to TRUE. If this is not done, HyperTalk will not proceed with the disposal of the window.

HyperTalk tries to close all open external windows when the user quits HyperCard. If any windows refuse to close at that time, HyperCard will not quit.

Do not use any of the following Toolbox routines to close external windows: `CloseWindow`, `DisposeWindow`, `CloseDialog`, or `DisposeDialog`.

Any XCMD can close any external window created with the `NewXWindow` or `GetNewXWindow` calls by means of a call to `CloseXWindow`. `CloseXWindow` has no effect on windows that were not created by means of a call to `NewXWindow` or `GetNewXWindow`.

You can also close an external window from a script with the form `send "close" to window` *windowName* or with the call `SendWindowMessage ("`*windowName*`","close")` from your XCMD.

Window utilities

```
PROCEDURE HideHCPalettes(paramPtr: XCmdPtr);
```

`HideHCPalettes` hides all of the HyperCard built-in palettes (the Tools palette, the Patterns palette, the FatBits window, and the Message box).

```
PROCEDURE RegisterXWMenu(paramPtr: XCmdPtr; window: WindowPtr;
                         menu: MenuHandle; registering: BOOLEAN);
```

`RegisterXWMenu` is useful only to XCMDs that manage external windows. `RegisterXWMenu` informs HyperCard that the given menu is meant for use with the external window in `window`. When an item in the menu is chosen by the user or specified as the menu item parameter to the `doMenu` command, the XCMD that registered the menu chosen receives a menu event (`xMenuEvt`). Note that an XCMD can register one of the HyperCard menus, for example, the Font menu, in order to borrow the menu temporarily.

RegisterXWMenu does not change the menu bar. The XCMD must call the Menu Manager in order to insert or delete the menu and to redraw the menu bar. Once the menu has been registered, it remains the property of the XCMD until RegisterXWMenu is called again with registering set to FALSE.

◆ *Note:* HyperCard expects XCMD menus to have unique IDs but does not ensure that they do. You may employ the following function to find an unused menu ID.

```
FUNCTION UnusedMenuID: INTEGER;
VAR thisID: INTEGER;
     menuHndl: MenuHandle;
BEGIN
   thisID := 1023;
   REPEAT
      thisID := thisID + 1;
      menuHndl := GetMHandle(thisID);
   UNTIL menuHndl = NIL;
   UnusedMenuID := thisID;
END;
```

```
PROCEDURE SetXWIdleTime(paramPtr: XCmdPtr; window: WindowPtr; ticks: LongInt);
```

XCMDs that manage external windows can request idle time from HyperCard. Use SetXWIdleTime if your XCMD needs to perform a periodic action.

Until SetXWIdleTime is called by an XCMD, the XCMD does not receive periodic calls. A value greater than 0 for ticks represents the requested interval between periodic calls by HyperCard. HyperCard sends the XCMD a nullEvent message when making its periodic call. Call SetXWIdleTime only after your external window has received an xOpenEvt.

Whether HyperCard periodic calls occur as often as requested depends on whether HyperCard is currently performing a timing-critical or data-intensive operation.

To give up idle time, call SetXWIdleTime with a value of 0 for ticks.

◆ *Note:* nullEvent messages are sent in both the foreground and background under MultiFinder.

```
PROCEDURE ShowHCPalettes(paramPtr: XCmdPtr);
```

ShowHCPalettes reverses the effect of HideHCPalettes, showing all of the HyperCard palettes that were visible when HideHCPalettes was called to hide them.

```
PROCEDURE XWHasInterruptCode(paramPtr: XCmdPtr; window: WindowPtr;
                             haveCode: BOOLEAN);
```

When HyperCard creates an external window (by means of either the NewXWindow or GetNewXWindow callback), it is assumed that the code of the XCMD that manages it can be moved in memory when it is not executing.

If haveCode is TRUE, HyperCard never unlocks the relocatable block containing the XCMD code in memory. This allows ProcPtrs and other addresses within an XCMD's code to be preserved and valid at all times. XWHasInterruptCode should be used with extreme prudence and should be undone as soon as possible— HyperCard memory management can become seriously taxed if there are any locked blocks inconveniently located in its heap. In particular, it's impossible to open card windows at the full size of large cards when a nonrelocatable block is located too close to the bottom of the application heap.

The preferred method for an XCMD to manage procedure pointers passed to the Toolbox is to refresh them as needed rather than to call XWHasInterruptCode to ensure their validity. This method permits HyperCard to unlock the XCMD's code between invocations. An example of such an XCMD is a text editor with a custom clikLoop routine. For best results, such an XCMD should recalculate and refresh its procPtr every time the user clicks in the viewRect. For example,

```
CASE evt.what OF
  mouseDown:
    BEGIN
      GlobalToLocal(evt.where);
      hTE^^.clikLoop := @MyClikLoop; { an assembly language clik loop }
      TEClick(evt.where, FALSE, hTE);
    END;
```

An XCMD that uses this method will have no need to call XWHasInterruptCode.

An XCMD's code is always locked while it is actually executing. XWHasInterruptCode determines only whether the block of code can be unlocked when control is returned to HyperCard.

```
PROCEDURE XWAlwaysMoveHigh(paramPtr: XCmdPtr; window: WindowPtr;
                           moveHigh: BOOLEAN);
```

`XWAlwaysMoveHigh` tells HyperTalk to always move the external window's code high on the heap before locking it down and calling it. External windows that may allocate large amounts of memory or send card messages to do such operations as go to another card should use this call.

In normal XCMD and external window operations, HyperTalk makes a determination at the time the code is jumped to as to whether the code should be moved high in the application heap before being locked down. Currently, this is done whenever an XCMD is called and when an external window is given the following events: `xOpenEvt`, `xMenuEvt`, `mouseDown`, and `keyDown`. All other events are assumed to rarely cause memory allocation or message sending; therefore, HyperTalk can speed up the process of calling and returning from an external window. However, some external windows may allocate memory on such things as an `xSetPropEvt` event. For example, a picture XCMD could be asked to change information about the currently displayed PICT and that may cause memory to be allocated. A good rule of thumb is: if your external window is not called repeatedly in a script and you might allocate memory on events other than the aforementioned events, call this with `moveHigh = TRUE`.

```
PROCEDURE XWAllowReEntrancy(paramPtr: XCmdPtr; window: WindowPtr;
                            allowSysEvts: BOOLEAN; allowHCEvts: BOOLEAN);
```

`XWAllowReEntrancy` allows an XCMD to tell HyperCard whether it is equipped to receive events (either system events or HyperCard-generated events) in a reentrant fashion. Either of the Booleans, `allowSysEvts` or `allowHCEvts`, can be set to TRUE or FALSE to enable or disable (respectively) this behavior.

In certain situations, it is possible for an external window to generate events for itself. In other words, calling the Window Manager's `InvalRect` routine on a portion of the window could create an `updateEvt` event for the window.

While running scripts, HyperTalk periodically checks the Event Manager to see if update, activate, or MultiFinder events are pending. This is how HyperCard processes events in the background. In the course of these checks, the Window Manager may report an update event for the external window if it notices that any or all of the window is invalid. The problem arises from external windows that use `SendCardMessage`, `SendHCMessage`, `EvalExpr`, or any other HyperTalk callback that can run a script. `XWAllowReentrancy` was implemented to allow external windows to perform asynchronous handling of certain events. For example, if a `SendCardMessage` callback causes a lengthy script to be executed, the external window may want to be notified that the user switched out of HyperCard under MultiFinder. The default is for HyperTalk to not allow these reentrant calls to take place. Unlike system events, HyperTalk events are not queued, so unreceived events are ignored.

- *Note:* An external window should only call `XWAllowReEnt` at the end of its response to an `xOpenEvt` event. The first event that is sent to an external window is its `xOpenEvt` event.

Because `xGetPropEvt` and `xSetPropEvt` are events, if an external window has not called `XWAllowReEntrancy(paramPtr,xxxx,TRUE)`, the scenario described below results in the external window never receiving its `xSetPropEvt` for its `visible` property:

```
SendCardMessage(paramPtr,'go next card');
    on closeCard
        hide window "fred" -- this call will fail
    end closeCard
```

The `xSetPropEvt` causes the external window to be reentered, so HyperTalk skips the event and continues. If your external window needs to communicate with running scripts, make sure that you allow it to be reentrant.

Writing completely reentrant code is difficult, and some development systems may not support reentrancy correctly. One tip is to make sure that every time an external window receives an event, it should completely save and restore its state. A good idea is to use the external window's `refCon` field to store a handle to all the window's state information. Beware of the use of global variables in stand-alone code resources. See Macintosh Technical Note #256 for more information.

Text editing utilities

```
PROCEDURE BeginXWEdit(paramPtr: XCmdPtr; window: WindowPtr);
```

`BeginXWEdit` registers an external window as the current editing environment. It can be called only by the XCMD that manages the window. Once it is called, HyperCard redirects all keystrokes to the XCMD, with the exception of Command-key combinations recognized by the Menu Manger as the equivalent of menu items. In addition, HyperCard passes events to the XCMD that correspond to the first five commands in the Edit menu whenever these commands are chosen. The XCMD still receives all of the other events pertaining to its window, including `nullEvents` if it has requested them.

Once the XCMD has registered itself as the current editing environment, it receives an `xGiveUpEdit` event from HyperCard when the user performs an action that activates a different editing environment, such as a click in the Message box or in an unlocked field. When this happens, the XCMD should deactivate its editable area as appropriate, just as it does when its window is deactivated.

```
PROCEDURE EndXWEdit(paramPtr: XCmdPtr; window: WindowPtr);
```

`EndXWEdit` informs HyperCard that an XCMD no longer wants to receive keystrokes and edit events for its external window. Call `EndXWEdit` before an external window that is an editing environment is closed via a call to `DisposeXWindow`.

```
FUNCTION HCWordBreakProc(paramPtr: XCmdPtr): ProcPtr;
```

`HCWordBreakProc` returns a procedure pointer to the HyperCard built-in word-break routine used for text in fields, scripts, and the Message box. Script Editors and other text-editing XCMDs can use this address in the `wordBreak` field of a `TextEdit` record.

```
PROCEDURE PrintTEHandle(paramPtr: XCmdPtr; hTE: TEHandle; header: StringPtr);
```

Given a handle to a `TextEdit` record in `hTE`, `PrintTEHandle` displays a print job dialog box and prints the record using the font, size, and style information contained within it. `PrintTEHandle` works for both old- and new-style edit records.

```
PROCEDURE FormatScript(paramPtr: XCmdPtr; scriptHndl: Handle;
                       VAR insertionPoint: LongInt;
                       quickFormat: BOOLEAN);
```

`FormatScript` reformats the script contained in the zero-terminated string to which `scriptHndl` is a handle. Call `ZeroTermHandle` before calling `FormatScript` if the text is not zero-terminated.

If `quickFormat` is `TRUE`, only the handler containing the insertion point is actually formatted. The offset into the text passed in `insertionPoint` is adjusted as necessary to reflect the same position within the text both before and after formatting. An XCMD that uses TextEdit should pass `TEHandle^^.selStart` for this parameter, then call `TESetSelection` after `FormatScript` returns. `InsertionPoint` is also used when `quickFormat` is `TRUE` to determine which handler within the script should be formatted.

- *Note:* using `EvalExpr` to get the script of an object formats it automatically.

```
FUNCTION GetCheckPoints(paramPtr: XCmdPtr): CheckPtHandle;
```

`GetCheckPoints` returns a handle to the cached checkpoints for the window's script. The caller must dispose of the handle. If there are no cached checkpoints, `GetCheckPoints` returns `NIL`.

The structure of a `CheckPtHandle` is as follows:

```
CheckPtHandle = ^CheckPtPtr;
CheckPtPtr = ^CheckPts;
CheckPts  = RECORD
              checks: ARRAY[1..16] OF INTEGER;
            END;
```

```
PROCEDURE SaveXWScript(paramPtr: XCmdPtr; scriptText: Handle);
```

Given a zero-terminated handle to a script in `scriptText`, `SaveXWScript` saves the script to disk. Call `ZeroTermHandle` before calling `SaveXWScript` if the text is not zero-terminated.

This callback is reserved for use by XCMDs that implement script editors. It fails if the XCMD has not been called by HyperCard as the current script editor. `SaveXWScript` also fails if the text of the script exceeds 32 KB.

◆ *Note:* An XCMD can set the script of any object using the following example code:

```
SetGlobal(paramPtr,'tempGlobal',scriptHndl);
SendCardMessage(paramPtr,CONCAT('set the script of',objectName,'to
        tempGlobal'));
SendCardMessage(paramPtr,'put empty into tempGlobal');
```

```
PROCEDURE SetCheckPoints(paramPtr: XCmdPtr; checkLines:
                         CheckPtHandle);
```

`SetCheckPoints` sets the checkpoints of a script to the array of lines in `checkLines`.

```
PROCEDURE GetObjectName(paramPtr: XCmdPtr; object: XTalkObjectPtr;
                        VAR objName: Str255);
```

Given an `XTalkObjectPtr` as input, this procedure returns the name of the object in `objName`.

```
PROCEDURE GetObjectScript(paramPtr: XCmdPtr; object: XTalkObjectPtr;
                          VAR scriptHndl: Handle);
```

Given an `XTalkObjectPtr` as input, `GetObjectScript` returns the script of the object in `scriptHndl`.

```
PROCEDURE SetObjectScript(paramPtr: XCmdPtr; object: XTalkObjectPtr;
                          scriptHndl: Handle);
```

Given an `XTalkObjectPtr` and a handle to a script as input, `SetObjectScript` sets the script of the object pointed to by `XTalkObjectPtr` to the script in `scriptHndl`.

Variable watcher support

```
PROCEDURE CountHandlers(paramPtr: XCmdPtr; VAR handlerCount:
                                    INTEGER);
```

`CountHandlers` returns the number of running handlers in `handlerCount`. If `handlerCount` is 0, then no handlers are currently running and the only variables that exist are global variables.

```
PROCEDURE GetHandlerInfo(paramPtr: XCmdPtr; handlerNum: INTEGER;
                                    VAR handlerName: Str255; VAR objectName:
                                    Str255; VAR varCount: INTEGER);
```

`GetHandlerInfo` returns information about handlers. If `handlerNum` is `1`, `GetHandlerInfo` returns information about the current handler, if there is one. If `handlerNum` is greater than `1`, `GetHandlerInfo` returns information about a handler that has a pending call to another handler. For example, if `handlerNum` is `2`, the information returned is for the handler that called the current handler. If there is no running handler, passing `0` for `handlerNum` yields information about the global variables.

```
PROCEDURE GetVarInfo(paramPtr: XCmdPtr; handlerNum: INTEGER; varNum:
                                INTEGER; VAR varName: Str255; VAR isGlobal:
                                BOOLEAN; VAR varValue: Str255; varHndl: Handle);
```

Given the variable number `varNum` in active handler number `handlerNum`, `GetVarInfo` returns information about the variable. If `varHndl` is `NIL`, a truncated version of the contents of the variable, limited to 255 characters, is returned in `varValue`. If `varHndl` is a valid handle, the entire contents of the variable are copied into that handle, which is resized as necessary. If there is no running handler, passing 0 for `handlerNum` returns information about the global variables.

```
PROCEDURE SetVarValue(paramPtr: XCmdPtr; handlerNum: INTEGER; varNum:
                                    INTEGER; varHndl: Handle);
```

`SetVarValue` sets the value of variable number `varNum` of handler `handlerNum` to the zero-terminated string passed in `varHndl`.

- *Note:* Changing the value of a variable other than the global variables (`handlerNum` 0) or the variables local to the current handler (`handlerNum` 1) does not affect the handler until it is once again current.

Debugger support

```
PROCEDURE AbortScript(paramPtr: XCmdPtr);
```

`AbortScript` cancels the currently executing handlers. The effect is the same as pressing Command-period or choosing Abort from the HyperCard built-in Debugger menu.

◆ *Note:* An XCMD that calls `AbortScript` is not itself aborted. It exits normally.

```
PROCEDURE GoScript(paramPtr: XCmdPtr);
```

`GoScript` exits the debugger, closes all temporary debugger windows, and continues normal execution of scripts.

```
PROCEDURE StepScript(paramPtr: XCmdPtr; stepInto: BOOLEAN);
```

`StepScript` creates a temporary checkpoint after the currently executing line. If `stepInto` is `TRUE`, the checkpoint is created at the next line of HyperTalk to be executed, even if it belongs to another handler. If `stepInto` is `FALSE`, the checkpoint is created at the next line of the current handler. `StepScript` causes the debugger to step out of a handler if the current line is its last.

```
FUNCTION GetStackCrawl(paramPtr: XCmdPtr): Handle;
```

`GetStackCrawl` returns a zero-terminated handle to the chain of callers (indented) as in the Message Watcher.

```
PROCEDURE TraceScript(paramPtr: XCmdPtr; traceInto: BOOLEAN);
```

`TraceScript` is similar to `StepScript` above, except that execution continues after the line is hilighted by the debugger. `TraceScript` respects the new HyperTalk property `traceDelay`.

External windows

The HyperCard XCMD interface includes support for external windows. The HyperCard built-in external windows are the script editor and the debugger tools Variable Watcher and Message Watcher, which are all described earlier in this book. All of these windows and their properties can be controlled through scripts. You can write your own XCMDs to replace any one of the built-in HyperCard external windows. You can also write XCMDs that provide any kind of useful application within a window, such as custom text editors, color picture editors, video overlay windows, and so on.

External windows are an extension of external commands. Two new callbacks, `NewXWindow` and `GetNewXWindow`, direct HyperCard to create a new external window. When an XCMD executes either of these callbacks, HyperCard creates a new window and saves with it a reference to the XCMD that made the call. Then, whenever HyperCard receives an event from the Toolbox Event Manager, it first determines whether the event pertains to an external window. If so, HyperCard calls the XCMD that created the window with arguments that allow it to handle the event. Otherwise, HyperCard handles the event itself.

Whenever an XCMD is called by HyperCard, it receives a pointer to an `XCMDBlock` parameter block. See "Parameter Block Data Structure," earlier in this chapter, for the Pascal parameter block structure.

When HyperCard calls an XCMD to handle an event for an external window, some of the fields of `XCMDBlock` have new values. The `paramCount` field is set to `-1`, indicating that the XCMD has been called to handle an event. The first parameter, `params[1]`, is a pointer to an `XWEventInfo` block, defined as follows:

```
XWEventInfoPtr = ^XWEventInfo;
XWEventInfo    = RECORD
                   event:       EventRecord;
                   eventWindow: WindowPtr;
                   eventParams: ARRAY[1..9] OF LongInt;
                   eventResult: Handle;
                 END;
```

Therefore, an XCMD that manages an external window can be structured as follows:

```
IF paramPtr^.paramCount >= 0 THEN CreateMyWindow
ELSE
  BEGIN
```

```
    WITH XWEventInfoPtr(paramPtr^.params[1])^ DO
      BEGIN
        myEvent := event;
        myWindow := eventWindow;
      END;
    SetPort(myWindow);
    CASE myEvent.what OF
      mouseDown: DoMouse;
      etc...
    END;
  END;
```

When an XCMD has been called to handle an event, it has full access to callback routines, just as it does when invoked from a HyperTalk script.

Events in external windows

HyperCard automatically sends most standard Macintosh events to XCMDs that manage external windows. These include the following events:

`nullEvent`	(See `SetXWIdleTime`.)
`mouseDown`	
`updateEvt`	
`activateEvt`	
`app4Evt`	(Palettes should hide themselves when suspended and show themselves when resumed.)

Keyboard events (`keyDown` and `autoKey`) are delivered to the XCMD only if it has registered itself as the active editing environment with the `BeginXWEdit` callback. All keystrokes, with the exception of Command-key combinations recognized by the Menu Manager as equivalents of menu items, are sent to an XCMD when it is the active editing environment.

In addition, XCMDs that manage windows receive messages specific to HyperCard. These are delivered in the same `EventRecord` data structure used for standard Macintosh events, with the `what` field set to one of the following constants:

```
xOpenEvt          = 1000;     { the first event creation }
xCloseEvt         = 1001;     { window is being asked to close }
xGiveUpEditEvt    = 1002;     { you are losing Edit... }
```

```
xGiveUpSoundEvt      = 1003;        { a request for the sound channel }
xHidePalettesEvt     = 1004;        { someone called HideHCPalettes }
xShowPalettesEvt     = 1005;        { someone called ShowHCPalettes }
xEditUndo            = 1100;        { Edit—Undo }
xEditCut             = 1102;        { Edit—Cut }
xEditCopy            = 1103;        { Edit—Copy }
xEditPaste           = 1104;        { Edit—Paste }
xEditClear           = 1105;        { Edit—Clear }
xSendEvt             = 1200;        { script has sent you a message (text) }
xSetPropEvt          = 1201;        { set a window property }
xGetPropEvt          = 1202;        { get a window property }
xCursorWithin        = 1300;        { cursor is within the window }
xMenuEvt             = 1400;        { user selected an item in your menu }
xMBarClickedEvt      = 1401;        { menu is to be shown--update items }
xShowWatchInfoEvt    = 1501;        { for variable and message watchers }
xScriptErrorEvt      = 1502;        { place the insertion point }
xDebugErrorEvt       = 1503;        { user clicked "Debug" at a complaint }
xDebugStepEvt        = 1504;        { hilite the line }
xDebugTraceEvt       = 1505;        { same as step but tracing }
xDebugFinishedEvt    = 1506;        { script ended }
```

Handling Events

Many of the events above require special attention. Many are specific to the debugger or other debugging tools. Here is a summary of all the events and their appropriate behavior:

`nullEvent` — If an external window requires idle time and registers itself as such using the `SetXWIdleTime` callback, it is called periodically with a `nullEvent`.

`mouseDown`
`updateEvt`
`app4Evt` — No special considerations are necessary for these events.

`activateEvt`

For external windows in the document layer (the same layer as the card window), activate/deactivate events are dispatched in the same manner as the Toolbox Event Manager. However, external windows in the floating layer *never* receive activate or deactivate events. Windows in the floating layer should always appear "active". If your external window requires more modality, then place it in the document layer.

`keyDown`
`autoKey`

If an external window has become the active editing window by calling `BeginXWEdit`, it receives all keystrokes as standard Macintosh events.

`xOpenEvt`

After the XCMD which created the external windows completes execution, HyperTalk sends an `xOpenEvt` to the new external window. This is always the first event an external window receives.

`xCloseEvt`

HyperTalk sends your external window an `xCloseEvt` when your window has called `CloseXWindow`, when another external window or XCMD has called `CloseXWindow` using your window's `WindowPtr`, or when the user has used the new `close window "windowName"` command. The external window should not dispose of any of its data until receiving this event. If the external window sets `passFlag` to `TRUE`, the window is disposed. If not, the window is left open. Refer to the discussion of the `CloseXWindow` callback for more information.

`xGiveUpEditEvt`

When an external window calls `BeginXWEdit`, it will receive keystrokes and edit menu commands until the user clicks in the card window or activates some other editing window (for example, the Message box). HyperTalk signals the current editor with an `xGiveUpEditEvt` event just before it activates the other editor.

`xGiveUpSoundEvt`	In the event that one external window requests the sound channel (with `BeginXSound`) while another external window has the channel, the current owner is given an `xGiveUpSoundEvt`. If the external window doesn't set `passFlag` to true, the other external window's callback returns with the result code set to `xresFail`. If an XCMD owns the sound channel, it can't be notified of the second request, so that request will fail. The only exception to this is the HyperCard patch to `_SysBeep`; if an XCMD or external window owns the sound channel and someone calls `SysBeep`, HyperCard flashes the menu bar rather than calling the XCMD.
`xHidePalettesEvt`	If another external window uses the `HideHCPalettes` callback, all external windows in the floating layer receive an `xHidePalettesEvt`. An example of this is the built-in Script Editor; it calls `HideHCPalettes` when the user edits a script.
`xShowPalettesEvt`	If another external window uses the `ShowHCPalettes` callback, all external windows in the floating layer receive an `xShowPalettesEvt`.
`xEditUndo`	While an external window is the editing environment, it receives this event, which corresponds to the Undo command in the HyperCard built-in Edit menu.
`xEditCut`	While an external window is the editing environment, it receives this event, which corresponds to the Cut command in the HyperCard built-in Edit menu.
`xEditCopy`	While an external window is the editing environment, it receives this event, which corresponds to the Copy command in the HyperCard built-in Edit menu.
`xEditPaste`	While an external window is the editing environment, it receives this event, which corresponds to the Paste command in the HyperCard built-in Edit menu.
`xEditClear`	While an external window is the editing environment, it receives this event, which corresponds to the Clear command in the HyperCard built-in Edit menu.

`xSendEvt`	When a user issues the `send` *"message"* `to window` *"windowName"* command, or when an XCMD or external window issues the `SendWindowMessage` callback, the external command receives an `xSendEvt` event. When this event is received, `eventParams[1]` contains a pointer to a Pascal string (`Str255`) containing the name of the message. This is done for speed purposes and to expedite using the `StringEqual` callback to index through the commands your window supports.
`xGetPropEvt`	HyperTalk contains new extensible syntax for getting properties of external windows. The syntax is `get` *property* `of window "`*windowName*`"`
`xSetPropEvt`	HyperTalk contains new extensible syntax for setting properties of external windows. The syntax is `set` *property* `of window "`*windowName*`" to` *propertyValue*

HyperTalk has two built-in properties of external windows: `loc` and `visible`. If an external window doesn't do anything special in response to being moved, shown, or hidden, it can set `passFlag` to TRUE in response to both `xSetPropEvt` and `xGetPropEvt` and HyperTalk handles the request. If a property is requested other than the two built-in ones and the external window passes the event, HyperTalk displays an error dialog box to the user.

If the external window wishes to respond specially to these requests, or if it has additional properties it wants to support, it can directly handle all HyperTalk requests. In both `xSetPropEvt` and `xGetPropEvt`, `eventParams[1]` contains a pointer to a Pascal string (`Str255`) containing the name of the property. In the case of a `xSetPropEvt`, `eventParams[2]` contains a handle to the property value. In the case of an `xGetPropEvt` event, the external window must return a handle in the `eventResult` field with the value of the property requested.

HyperTalk gets and sets the `visible` property to translate the `hide` and `show` commands. In the event the user says `hide window "Variable Watcher"`, HyperTalk informs the external window by giving it an `xSetPropEvt` event with the property `visible` and the value `false`.

For an example of the ability of external windows to use properties, try these properties of the Message Watcher window: `loc`, `visible`, `hideIdle`, `hideUnused`, `text`, and `lastLine` (set only). The Variable Watcher supports the following properties: `loc`, `visible`, `hBarLoc`, `vBarLoc`, and `rect`.

There are four new callbacks to aid in getting and setting properties: `PointToStr`, `RectToStr`, `StrToPoint`, and `StrToRect`.

`xCursorWithin`	This event is sent when the cursor is over any part of the external window. If the external window sets `passFlag` to `TRUE`, HyperCard sets the cursor to an arrow.
`xMenuEvt`	When an external window has used `RegisterXWMenu` for one or more menus in the HyperCard menu bar, HyperTalk notifies it of an item being chosen by sending it an `xMenuEvt` event. This holds true for Command-key equivalents as well. When the event is received, `eventParams[1]` is the menu ID and `eventParams[2]` is the menu item.
`xMBarClickedEvt`	If an external window has registered one or more menus in the menu bar, HyperTalk sends it an `xMBarClickedEvt` event just before it calls the Menu Manager's `MenuSelect` or `MenuKey` routines. This is to allow the external window to adjust its menus just before the user sees them.
`xShowWatchInfoEvt`	This event is sent to the Message Watcher and Variable Watcher whenever it is appropriate. For the Message Watcher, `eventParams[1]` contains a handle to the current message. For the Variable Watcher, there are no arguments sent.
`xScriptErrorEvt`	This event is sent to the current script editor when HyperTalk displays the "Debug, Script, or Cancel" alert and the user clicks Script. `EventParams[1]` contains the line number on which the error occurred.

`xDebugErrorEvt`
`xDebugStepEvt`
`xDebugTraceEvt`
	This event is sent to the current debugger window.

`xDebugFinishedEvt` This event is sent to all external windows when the user chooses the Go or Abort commands in the Debugger menu.

Closing an external window

External windows should *only* dispose of their private data structures in response to an `xCloseEvt` event. Whenever an XCMD calls `CloseXWindow`, it should not dispose of any of its internal information until it receives its `xCloseEvt` event.

When an `xCloseEvt` event is sent, the XCMD must set `passFlag` to TRUE if it actually wants to close. If it does not set `passFlag`, it is telling HyperTalk that it wants to cancel the close operation. This is useful for editors that put up a "Save this document?" dialog box that includes a Cancel button.

Special XCmdBlock values

Because special XCMDs like the Message Watcher and script editor look like normal XCMDs, there has to be some defense against a user typing `AppleEditor` in the Message box and launching the script editor. Therefore, the special-case XCMDs are given special parameter blocks when they are opened by HyperTalk.

Message Watcher

When HyperTalk calls the Message Watcher to initialize itself, it sets `paramPtr^.paramCount` to `xMessageWatcherID (-2)`. There are no parameters sent. The Message Watcher is called at startup time and is always present. If you switch to another message watcher (using `set the messageWatcher to "MyWatcher"`), that watcher is loaded immediately. The normal behavior of a message watcher is that it should be inactive when it is invisible.

Variable Watcher

When HyperTalk calls the Variable Watcher to initialize itself, it sets `paramPtr^.paramCount` to `xVariableWatcherID (-3)`. There are no parameters sent. The Variable Watcher is called at startup time and is always present. If you switch to another variable watcher (using `set the variableWatcher to "`*MyWatcher*`"`), that watcher is loaded immediately. The normal behavior of a variable watcher is that it should be inactive when it is invisible.

Script editor

When HyperTalk calls a script editor to initialize itself, it sets `paramPtr^.paramCount` to `xScriptEditorID (-4)`. There are three parameters sent. `Params[1]` is a zero-terminated handle to the script of the object (unindented). `Params[2]` is a pointer to a Pascal string (`Str255`) containing the name of the window as proposed by HyperTalk (for example, `script of card button id 4 = "Fred"`). `Params[3]` is a pointer to the `XTalkObject` structure describing the object to be edited. (See the next section.)

◆ *Note:* There can be multiple script editors open at once, all sharing the same copy of the XCMD resource in memory. For this reason, XCMDs must not write to their own own code, or serious problems can result.

Debugger

A debugger window is initialized the same as a script editor, with the exception that the `paramPtr^.paramCount` is set to `xDebuggerID (-5)`.

XTalkObject

In order to increase a script editor's flexibility, it can communicate with HyperTalk using a special data structure called `XTalkObject`. Each time a script editor is opened, it is passed a pointer to this structure describing the object whose script is being edited. However, any XCMD can use it as long as the relevant fields are filled in correctly.

```
XTalkObjectPtr = ^XTalkObject;
XTalkObject = RECORD
                objectKind:  INTEGER;    { stack, bkgnd, card, field,
                                           or button }
                stackNum:    LongInt;    { reference number of the
                                           source stack }
                bkgndID:     LongInt;
                cardID:      LongInt;
                buttonID:    LongInt;
                fieldID:     LongInt;
              END;
```

`objectKind` The type of object: `stackObj`, `bkgndObj`, `cardObj`, `fieldObj`, and `buttonObj`.

`stackNum` The reference number of the stack containing the object. Use `StackNameToNum` to get the `stacknum` of a stack when you know the name. Use `GetObjectName` to get the name of the stack if all you know is the number. If `XTalkObject.objectKind` = `stackObj`, this is the last relevant field.

`bkgndID` The ID of the background. If `XTalkObject.objectKind` = `bkgndObj`, this is the last relevant field. If the object is a card, this is the background to which the card belongs. If the object is a background field or button, this is the ID of the background to which the field or button belongs.

`cardID` The ID of the card. If `XTalkObject.objectKind` = `cardObj`, this is the last relevant field. If the object is a card field or button, this is the ID of the card to which the field or button belongs.

`buttonID` If the button is a card button, this value is negative, `ABS(buttonID)` is the ID of the button, and `cardID` is the owner of the button. If the button is a background button, this value is positive, `buttonID` is the ID of the button, and `bkgndID` is the owner of the button.

`fieldID` If the field is a card field, this value is negative, `ABS(fieldID)` is the ID of the field, and `cardID` is the owner of the field. If the field is a background field, this value is positive, `fieldID` is the ID of the field, and `bkgndID` is the owner of the field.

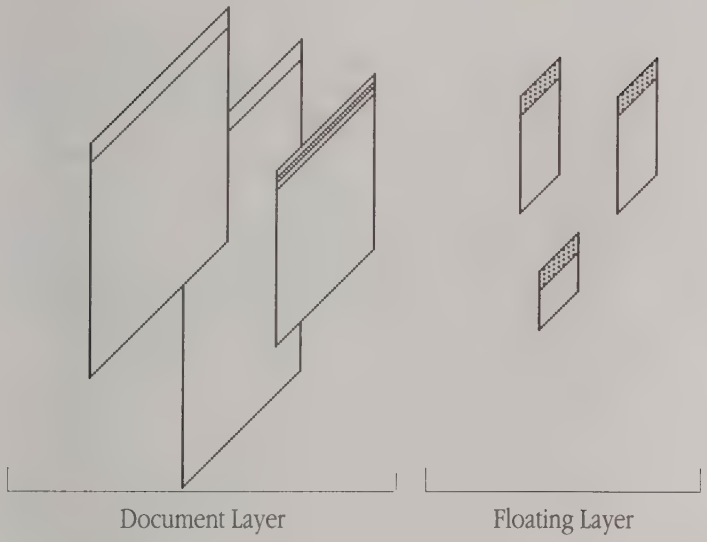
Window layer management

Within HyperCard, each window resides in one of two layers, the floating layer or the document layer, as shown in Figure A-2.

The floating layer is reserved for windows that have a single state, such as the HyperCard Tools palette, which is always active when it is visible. All of the HyperCard floating windows, including the Message box and the Tools palette, reside in this layer. Windows in this layer never receive activate events.

The rear layer is the document layer. Windows that have multiple states, active and inactive, such as HyperCard card windows and script editing windows, reside in this layer.

■ **Figure A-2** HyperCard window layers



If your window resides in the document layer and you wish to determine whether it's frontmost in its layer, use the `FrontDocWindow` callback. The Window Manager routines, such as `FindWindow` and `FrontWindow`, operate in HyperCard window environment. For example, `FrontWindow` returns a pointer to the frontmost window.

HyperCard automatically inserts new external windows in the proper layer according to type. (See the description of the `NewXWindow` callback.)

An XCMD can determine the layer in which its external windows reside by using the `floating` argument to `GetNewXWindow` and `NewXWindow`. If `floating` is TRUE, the window is placed at the front of the floating layer, otherwise it is placed at the front of the document layer.

```
FUNCTION NewXWindow(paramPtr: XCmdPtr; boundsRect: Rect; title: Str255;
                    visible: BOOLEAN; procID: INTEGER; color: BOOLEAN;
                    floating: BOOLEAN): WindowPtr;

FUNCTION GetNewXWindow(paramPtr: XCmdPtr; templateType: ResType;
                       templateID: INTEGER; color: BOOLEAN; floating:
                       BOOLEAN): WindowPtr;
```

Memory Management Tips

The Macintosh Memory Manager will allocate two different types of heap blocks: relocatable and nonrelocatable. Nonrelocatable blocks (`pointers`) are allocated as low as possible in the heap, and relocatable blocks (`handles`) are allocated wherever possible. A problem will arise if an XCMD or external window leaves a nonrelocatable block (or a locked relocatable block) allocated after it returns control to HyperCard and the user attempts to go to a stack that is larger than classic size (512 pixels wide and 342 pixels high). This condition is commonly called an island in the heap. Even though the HyperTalk `heapSpace` function may indicate lots of free memory, the user will be limited to classic card size, scrolling will be slower, and the user will be prevented from using the Painting tools.

Follow the guidelines below and life will be far richer for your XCMD and XFCN users:

Pointers? Nonrelocatable blocks are OK to use temporarily, but do not leave them allocated when the XCMD returns control to HyperCard.

Locked Handles? Locked relocatable blocks are also OK to use, but try to lock the block only when it is actually being accessed, then either dispose it or unlock it.

Permanently? If an XCMD requires a block to be locked permanently, it should call `NewHandle`, and if that call is successful, it should call `MoveHHi` before it calls `HLock`. Some more complex XCMDs may contain code meant to run at interrupt level or code (such as ROM patches) to which absolute addresses must be calculated. Once a relocatable block is locked, that block is just as immovable as a nonrelocatable block, but much nicer from the HyperCard user's perspective.

Callbacks? Also, be careful with callbacks such as `SendCardMessage` while blocks are locked in the heap, because these callbacks can cause HyperCard to change stacks if the message issues a `go` command.

Flash: an example XCMD

A simple example external command included with HyperCard is `flash`, which inverts the screen display (changes the black pixels to white and vice versa) a specified number of times. A version of `flash` written and compiled in MPW Pascal has already been attached to the HyperCard application file (that is, to HyperCard itself).

`Flash` is invoked from HyperCard just like a HyperTalk command. That is, you send the message `flash` to HyperCard from the Message box or from an executing script. The flash message takes one parameter: an integer. The `flash` XCMD inverts the screen display twice that many times. For example, the following handler, in response to a `mouseUp` message, sends the `flash` message and its parameter. When the message reaches HyperCard, it invokes the `flash` external command, which inverts the screen display 20 times:

```
on mouseUp
       flash 10
end mouseUp
```

The screen display flashes (is inverted and inverted back again) 10 times.

Flash listing in MPW Pascal

Here's the Pascal listing for `flash`:

```
(*
 *  Flash.p    - A sample HyperCard XCMD to highlight the screen
 *             - Copyright Apple Computer, Inc. 1987-1989.
 *             - All Rights Reserved.
 *
 *  Build instructions:
 *
 *  Pascal Flash.p -o Flash.p.o
 *  Link Flash.p.o -sg Flash -rt XCMD=0 -m ENTRYPOINT -o StackName
 *
 *)

{$R-}
```

```
{$S Flash }   { Segment name must be same as command name }
(*
 *  DummyUnit is what HyperTalk jumps to when running the XCMD.
 *  Also note that XCMDs do not currently support their own A5 World,
 *  thus NO GLOBAL VARIABLES are allowed. If the link fails then that
 *  means the Pascal compiler generated A5-relative code. (This may
 *  happen if you try to use the Pascal libraries, for example.)
 *
 *)

UNIT DummyUnit;

INTERFACE

USES MemTypes, QuickDraw, HyperXCmd;

PROCEDURE EntryPoint(paramPtr: XCmdPtr);

IMPLEMENTATION

TYPE Str31 = String[31];

PROCEDURE Flash(paramPtr: XCmdPtr); FORWARD;

  PROCEDURE EntryPoint(paramPtr: XCmdPtr);
  BEGIN
    Flash(paramPtr);
  END;

  PROCEDURE Flash(paramPtr: XCmdPtr);
  VAR flashCount:       INTEGER;
    again:              INTEGER;
    port:               GrafPtr;
    str:                Str255;
    when:               LongInt;
```

```
    ticksPtr:           ^LongInt;

  {$I XCmdGlue.inc }
  BEGIN
    ZeroToPas(paramPtr^.params[1]^,str);        { first param is flash count }
    flashCount := StrToNum(str);
    GetPort(port);
    ticksPtr := Pointer($16A);

    IF (paramPtr^.paramCount <> 1) OR (flashCount < 1)
    THEN flashCount := 3;

    FOR again := 1 TO 2 * flashCount DO
      BEGIN
        when := ticksPtr^ + 4;
        InvertRect(port^.portRect);
        REPEAT UNTIL ticksPtr^ >= when;
      END;
  END;

END.
```

Flash listing in MPW C

Here's a version of `flash` written in MPW C:

```
/*
 *          Flash.c -       A sample HyperCard XCMD to highlight the screen
 *                  -       Copyright Apple Computer, Inc. 1987-1989.
 *                  -       All Rights Reserved.
 *
 *          Build instructions:
 *
 *          C Flash.c -o Flash.c.o
 *          Link Flash.c.o -sg CFlash -rt XCMD=5 -o StackName
 *
```

```
 */

#define __SEG__ CFlash /* Segment name must be the same as command name */

#include <HyperXCmd.h> /* HT interface and #includes Types.h, Memory.h */
#include <QuickDraw.h>

pascal void   MacsBug() extern 0xA9FF; /* useful for debugging */
/*
 *   Your routine MUST be the first code that is generated in the file, as
 *   HyperTalk simply JSRs to the start of the XCMD segment in memory.
 * Therefore the XCmdGlue.c file must be included after the main routine,
 *   being CFlash in this sample XCMD. Also note that XCMDs do not currently
 *   support their own A5 World, thus NO GLOBAL VARIABLES are allowed.
 * If the link fails then that means the C compiler generated A5-relative
 *   code. (This happens if you try to use the C libraries or use strings
 *   in the code. Use a STR resource instead.)
 *
 */

pascal void CFlash(paramPtr)
     XCmdBlockPtr paramPtr;
{
     short flashCount,again;
     GrafPtr port;
     Str255 str;

     ZeroToPas(paramPtr,*(paramPtr->params[0]),&str); /*get flash count */
     flashCount = StrToNum(paramPtr,&str); /* convert to num */
     if (paramPtr->paramCount != 1) flashCount = 3; /* default if no param */
     if (flashCount < 1) flashCount = 3; /* must be positive */
     GetPort(&port);
     for (again = 1; again <= flashCount; again++) {
         InvertRect(&port->portRect);
         InvertRect(&port->portRect);
     }
}
```

```
#include <XCmdGlue.c>        /* C routines for HyperCard callbacks */
```

Flash listing in 68000 assembly language

Here's the 68000 assembly language listing for `flash`:

```
*
* Flash.a    - A sample HyperCard XCMD in 68000 Assembly
*            - Copyright Apple Computer, Inc. 1988-1989.
*            - All Rights Reserved.
*
* Build Instructions:
*
*            Asm Flash.a -o Flash.a.o
*            Link Flash.a.o -sg AFlash -rt XCMD=7 -o StackName
*
*

        INCLUDE 'QuickEqu.a'
        INCLUDE 'Traps.a'

          SEG 'AFlash'                  ; Segname must be same as command name

AFlash    PROC                          ; uses a0,a1,d1
          link a6,#-4
          move.l d4,-(sp)               ; save
          move.l 8(a6),a0               ; get paramPtr in a temp reg
          move.l 2(a0),a1               ; get handle to flashCount (as c string)
          move.l (a1),a1                ; deref
          move.w #3,d4                  ; StrToNum default result

@1        move.b (a1)+,d1               ; get a char
          cmp.b #'0',d1                 ; test for a number
          blt.s @2                      ; less than valid
          cmp.b #'9',d1
          bgt.s @2                      ; greater than valid
```

```
        and.w #$000F,d1             ; mask to value of legal char
        move.w d1,d4                ; stick value into result

@2      pea  -4(a6)                 ; var result of GetPort
        _GetPort
        bra.s @4                    ; get into DBRA loop

@3      move.l -4(a6),a0            ; get port
        pea  portRect(a0)           ; address of portRect
        _InverRect
        move.l -4(a6),a0            ; get port
        pea  portRect(a0)           ; address of portRect
        _InverRect
@4      dbra d4,@3

        move.l (sp)+,d4             ; restore
        unlk a6

        move.l (sp)+,a0             ; rts Pascal style
        add.l #4,a7
        jmp  (a0)

        END
```

Appendix B **Constants**

This chapter describes the HyperTalk built-in constants. A **constant** is a named value that never changes. It's different from a variable because you can't change it, and it's different from a literal because it does not require quotation marks.

The values of some constants are the string of characters making up the name, while others are different. In some cases, it's more convenient to use a constant (such as `pi`) in place of a long string (such as 3.14159265358979323846). In other cases, it's more convenient to use a constant (such as `formFeed`) because the only other way to enter that character is with the `numToChar` function, requiring that you know the ASCII number of the character (as in `the numToChar of 12`).

You can't name a variable the same as any built-in constant; if you try, HyperCard displays an error dialog box.

Table B-1 is a list of all the built-in constants in HyperTalk.

■ **Table B-1** HyperTalk constants

Constant name	Description
`down`	The value returned by the `commandKey`, `mouse`, `optionKey`, or `shiftKey` function when the named key (or button, in the case of `mouse`), is currently pressed. Its value is the same as the literal `"down"`.
`empty`	The null string, the same as the literal `""`.
`false`	The opposite of `true`; one of the states tested by the `if` control structure and one of the possible results of evaluation of a logical expression. Its value is the same as the literal `"false"`.
`formFeed`	The form feed character (ASCII 12), which starts a new page in some file formats.
`lineFeed`	The line feed character (ASCII 10), which starts a new line in some file formats.
`pi`	The mathematical value pi to 20 decimal places, denoting the ratio of the circumference of a circle to its diameter, represented by the number 3.14159265358979323846.

`quote`	The double quotation mark character. It is needed to build a string containing quotation marks because they are stripped out of the string when literals are evaluated:
	`put "george" into It -- quotation marks are not in It`
	`put quote & "george" & quote into It`
	`-- quotation marks are in It`

`return`	The return character (ASCII 13), which signifies the end of a HyperTalk statement.
`space`	The space character (ASCII 32), the same as the literal `" "`.
`tab`	The horizontal tab character (ASCII 9).
`true`	The opposite of `false`; one of the states tested by the `if` control structure and one of the possible results of evaluation of a logical expression. Its value is the same as the literal `"true"`.
`up`	The value returned by the `commandKey`, `mouse`, `optionKey`, and `shiftKey` functions when the named key (or button, in the case of `mouse`), is not currently pressed. Its value is the same as the literal `"up"`.
`zero..ten`	The numbers 0 through 10.

Appendix C Enhancing the Execution Speed of HyperCard

This chapter provides scripting hints and other techniques for getting the most performance out of HyperCard applications.

One of the key methods for increasing the speed of HyperCard is avoiding disk accesses whenever possible. If you remember a few good scripting techniques, you can keep disk access at a minimum. To avoid excessive disk accesses, do the following:

- Change stacks as seldom as possible.
- Use variables instead of fields for all operations.
- Refer to a remote card, rather than going there.

Other good scripting techniques that generally improve the performance of HyperCard are as follows:

- Migrate to XCMDs and XFCNs for highly repetitive tasks, such as sorting.
- Set `lockScreen` to `true` to avoid needless redrawing.
- Set `lockMessages` to `true` to save time during card-to-card data collection.
- Combine multiple messages.
- Take unnecessary code out of loops.
- Always use quoted literals.
- Use in-line statements rather than handler calls.
- Do complex calculations once.
- Watch overuse of variable references.
- Do visible work first.

Each of the techniques is descibed in the sections that follow.

Change stacks as seldom as possible

Changing stacks means going to a disk, hard disk, or CD-ROM to retrieve information and in the case of a floppy disk and hard disk, to store information. The disk accessing process takes more time that any other HyperCard operation.

When you need to read or write data from another stack, go the stack only once. Whenever possible, get or put everything you need at the same time.

Keep related information in the same stack. This includes data that you regularly sort, search, or move between. If you plan to use cards with substantially different appearances within the same application (a HyperCard application could consist of several stacks) and you need to cross boundaries frequently, use multiple backgrounds in the same stack rather than separate stacks.

However, there are exceptions to this rule. Because you might want to put the finished stack on a floppy disk for ease of backup and transportation, you need to keep an eye on the size of stacks as you build them. If you have too much related data to put on a single floppy, you have to use multiple stacks rather than one stack. Alternatively, you could put the final stack on a CD-ROM.

Another exception to the single stack rule is command operations. The HyperCard `sort` and `find` commands operate on the entire stack, and you may not want the entire stack sorted or searched.

Use variables, not fields, for operations

Whenever possible, do operations such as sorting, data collection, and calculations in variables rather than fields. HyperCard operations are much faster in variables.

Fields are for the display of data and for long-term storage; HyperCard keeps field information stored on disk and draws that information onto the screen. Variables, on the other hand, are in RAM and are for storage and manipulation of transient data—data that doesn't appear on the screen and is lost when you quit HyperCard. For example, collecting data on different cards into variables is approximately 50% faster than collecting the same data into a field.

Do calculations in variables to get faster results, for example, when adding the contents of a series of fields from different cards. Everything that goes into fields, including numbers, is converted to a string, regardless of its original format. The numeric content of variables, on the other hand, is stored in binary format, making calculation more precise and less time consuming.

Refer to a remote card rather than going there

Referring to a remote card is generally faster than going there, but is dependent on the number of fields referred to per card. For example, when collecting data from fewer than 10 fields, referring to a remote card in the currrent stack is faster than going to the card. On the other hand, when collecting data from 10 fields or more, it is faster to go to that card and then collect the data. For example, this script refers to a remote card containing data needed from the fields on that card. The script works in the current card of the current stack.

```
on collectGoodies
  put empty into FieldNum
  repeat for FieldNum = 1 to 8
    put line 2 in card field FieldNum of card 4 & return & line 3 in card¬
    field FieldNum of card 4 & return after field 3 of this card
  end repeat
end collectGoodies
```

If the data is needed from more than 10 fields on a remote card, the script might look like this:

```
on collectGoodies
  set lockMessages to true
  put empty into FieldNum
  push card
  lock screen
  go to card 4
  repeat for FieldNum = 1 to the number of bkgnd fields
    put line 2 in card field FieldNum of card 4 & return & line 3 in card¬
    field FieldNum of card 4 & return after field 3 of card 1
  end repeat
  unlock screen
  pop card
  set lockMessages to false
end collectGoodies
```

Migrate to XCMDs and XFCNs for repetitive tasks

No matter how efficient your HyperTalk code is, sorting 500 items is still going to take a long time. Repetitive time-intensive tasks are best handled with XCMDs and XFCNs. XCMDs and XFCNs have the same calling interface as any other HyperCard command, so are not any harder to use in a script.

If you do not have the programming experience in another high-level language needed to create an appropriate XCMD for your application, public domain and shareware XCMDs may exist that provide all of the required functions that your application needs. These XCMDs can be found through HyperCard user groups, user bulletin boards, and other information services.

Set lockScreen to true to avoid needless redrawing

Redrawing the screen takes time and it makes no sense to change the screen if all you are doing is going to another card to collect data. Set `lockSreen` to `true` (or use the `lock screen` command) while your scripts collect data by going from card to card and stack to stack.

See the second script example in the section "Refer to a Remote Card Rather Than Going There."

Set lockMessages to true during card-to-card data collection

The `lockMessages` command prevents HyperCard from sending the six open and close system messages associated with cards, backgrounds, and stacks. See Chapter 8, "System Messages," for more information about HyperCard system messages.

Combine multiple messages

Message sending is relatively time consuming because each message traverses the entire message-passing path. The techniques described here avoid excessive message sending and can save lots of time.

Operators are interpreted directly and don't incur the cost of message sending. Use of the operators `is in`, `is not in`, and `contains` are faster than using `offset()` because they are interpreted directly and are not sent as messages.

Calling a built-in function with `the` or `of` is faster than calling with "`()`" because "`()`" functions traverse the message path. Therefore, using `date()` is slower than using `the date`.

Use `put` to move values directly, rather than using `get` to store a value in `It` and then using `put` to move the value out of `It`. As an example,

```
get x
put It into y
```

is slower than

```
put x into y
```

Take unnecessary code out of loops

The reason for removing unnecessary code from loops is fairly obvious, but frequently overlooked. Shorter handlers with less lines of code take less time to run than longer handlers. This fact is magnified in loops. For example, a loop that comprises 6 lines and runs six iterations is equivalent to 25 lines of code. Every extra line in that six-iteration loop counts as an additional 6 lines. Unwrap your code where possible and avoid unecessary lines. For example, don't start a loop with the line `set the cursor to watch`, because you get that for free.

Use in-line statements rather than handler calls

It always takes time to get from one place to another and back again in any HyperCard application and script. To save time, put all of your code in-line. Create utility handlers where possible to avoid calling back to handlers in the main module. Putting a lot of your code in-line can, however, make your code hard to read. You should consider whether the increase in speed is worth having code that is hard to read. A good solution is to save most of the in-line code for repetitive time-consuming tasks that you haven't written XCMDs to handle.

Do complex calculations once

When you've figured something out, put the results in a variable, then refer to the variable. This also includes parameter values. Retrieve parameter values once rather than many times (assuming you know the values haven't changed). This technique results in the greatest time savings within loops.

Watch overuse of variable references

The speed increases seen by avoiding overuse of variable references are minimal, but it is good scripting technique. The line `add x to y` is slightly faster than `put x + y into y`, because the former has fewer variable references (two) than the latter (three).

Appendix D **Extended ASCII Table**

This appendix lists the character assignments for the 256 single-byte character values used by the Macintosh.

There are 256 possible 8-bit binary values, from 00000000 to 11111111. Of these, the first 128 (from 00000000 to 01111111) have been assigned to a standard set of characters and commands used in data processing and communication. These assignments form the ASCII character set. (*ASCII* stands for *American Standard Code for Information Interchange*.)

The remaining 128 binary values, those for which the most significant bit (first digit) is 1 instead of 0, are not assigned in the ASCII standard. Because they have higher numerical values than the first 128 characters, they are often referred to as high-ASCII characters.

This appendix lists all character values by their decimal equivalent.

Table D-1 lists the first 32 characters, the control characters, which have no printable-character representation, with the standard abbreviation for each and its meaning.

Table D-2 lists the remaining 224 character values with the characters to which they are assigned in the Macintosh Courier font.

■ **Table D-1** Control character assignments

Value	Name	Meaning	Value	Name	Meaning
0	NUL	Null	16	DLE	Data link escape
1	SOH	Start of heading	17	DC1	Device control 1
2	STX	Start of text	18	DC2	Device control 2
3	ETX	End of text	19	DC3	Device control 3
4	EOT	End of transmission	20	DC4	Device control 4
5	ENQ	Enquiry	21	NAK	Negative acknowledge
6	ACK	Acknowledge	22	SYN	Synchronous idle
7	BEL	Bell	23	ETB	End of transmission block
8	BS	Backspace	24	CAN	Cancel
9	HT	Horizontal tab	25	EM	End of medium
10	LF	Line feed	26	SUB	Substitute
11	VT	Vertical tab	27	ESC	Escape
12	FF	Form feed	28	FS	File separator
13	CR	Carriage return	29	GS	Group separator
14	SO	Shift out	30	RS	Record separator
15	SI	Shift in	31	US	Unit separator

■ **Table D-2** Character assignments in Macintosh Courier font

Value	Character	Value	Character	Value	Character	Value	Character
32	SPACE	61	=	90	Z	119	w
33	!	62	>	91	[	120	x
34	"	63	?	92	\	121	y
35	#	64	@	93	]	122	z
36	$	65	A	94	^	123	{
37	%	66	B	95	_	124	|
38	&	67	C	96	`	125	}
39	'	68	D	97	a	126	~
40	(	69	E	98	b	127	DEL
41	)	70	F	99	c	128	Ä
42	*	71	G	100	d	129	Å
43	+	72	H	101	e	130	Ç
44	,	73	I	102	f	131	É
45	-	74	J	103	g	132	Ñ
46	.	75	K	104	h	133	Ö
47	/	76	L	105	i	134	Ü
48	0	77	M	106	j	135	á
49	1	78	N	107	k	136	à
50	2	79	O	108	l	137	â
51	3	80	P	109	m	138	ä
52	4	81	Q	110	n	139	ã
53	5	82	R	111	o	140	å
54	6	83	S	112	p	141	ç
55	7	84	T	113	q	142	é
56	8	85	U	114	r	143	è
57	9	86	V	115	s	144	ê
58	:	87	W	116	t	145	ë
59	;	88	X	117	u	146	í
60	<	89	Y	118	v	147	ì

[continued]

■ **Table D-2** Character assignments in Macintosh Courier font [continued]

Value	Character	Value	Character	Value	Character	Value	Character
148	î	175	Ø	202	**	229	Â
149	ï	176	∞	203	À	230	Ê
150	ñ	177	±	204	Ã	231	Á
151	ó	178	≤	205	Õ	232	Ë
152	ò	179	≥	206	Œ	233	È
153	ô	180	¥	207	œ	234	Í
154	ö	181	µ	208	–	235	Î
155	õ	182	∂	209	—	236	Ï
156	ú	183	Σ	210	"	237	Ì
157	ù	184	Π	211	"	238	Ó
158	û	185	π	212	'	239	Ô
159	ü	186	∫	213	'	240	
160	†	187	ª	214	÷	241	Ò
161	°	188	º	215	◊	242	Ú
162	¢	189	Ω	216	ÿ	243	Û
163	£	190	æ	217	Ÿ	244	Ù
164	§	191	ø	218	⁄	245	ı
165	•	192	¿	219	¤	246	ˆ
166	¶	193	¡	220	‹	247	˜
167	ß	194	¬	221	›	248	¯
168	®	195	√	222	fi	249	˘
169	©	196	ƒ	223	fl	250	˙
170	™	197	≈	224	‡	251	°
171	´	198	Δ	225	·	252	¸
172	¨	199	«	226	‚	253	˝
173	≠	200	»	227	„	254	˛
174	Æ	201	...	228	‰	255	ˇ

** Stands for a nonbreaking space

Appendix E **Operator Precedence Table**

This appendix shows the order of precedence of HyperTalk operators. In a complex expression containing more than one operator, HyperTalk performs the operation indicated by operators with lower-numbered precedence before those with higher-numbered precedence. Operators of equal precedence are evaluated left-to-right, except for exponentiation, which goes right-to-left. If you use parentheses, HyperTalk evaluates the innermost parenthetical expression first.

Chapter 6 discusses expression evaluation.

■ **Table E-1** Operator precedence

Order	Operators	Type of operator
1	`( )`	Grouping
2	`-`	Minus sign for numbers
	`not`	Logical negation for Boolean values
	`there is a`	Comparison for HyperCard items
	`there is an`	Comparison for HyperCard items
	`there is not a`	Comparison for HyperCard items
	`there is not an`	Comparison for HyperCard items
	`within`	Boolean test for point within rectangle
3	`^`	Exponentiation for numbers
4	`* / div mod`	Multiplication and division for numbers
5	`+ -`	Addition and subtraction for numbers
6	`& &&`	Concatenation of text
7	`> < <= >= ≤ ≥`	Comparison for numbers or text
	`is in contains`	Comparison for text
	`is not in`	Comparison for text
	`is a`	Comparison for types
	`is an`	Comparison for types
	`is not a`	Comparison for types
	`is not an`	Comparison for types
8	`= is is not <> ≠`	Comparison for numbers or text
9	`and`	Logical for Boolean values
10	`or`	Logical for Boolean values

Appendix F **HyperCard Synonyms**

Table F-1 lists the alternative ways that HyperTalk terms can be used.

■ **Table F-1** HyperTalk synonyms

Synonym	Term		
abbr	abbrev	abbreviated	
bg	bkgnd	background	
bgs	bkgnds	backgrounds	
btn	button		
btns	buttons		
cd	card		
cds	cards		
char	character		
chars	characters		
fade	dissolve		
fld	field		
flds	fields		
grey	gray		
hilited	highlighted	highlited	hilighted
loc	location		
msg	message	message box	
msg watcher	message watcher		
mid	middle		
poly	polygon		
prev	previous		
rect	rectangle		
reg	regular		
sec	secs	seconds	
spray	spray can		
tick	ticks		

Appendix G **HyperCard Limits**

This appendix lists various minimum and maximum sizes and numbers of elements defined in HyperCard.

The maximum limits shown in this appendix are theoretical. Some of them are lower in practice. For example, HyperCard currently brings an entire card into memory at once, so the maximum size of a card is limited by available memory. It's possible that a card with a lot of text and long scripts, created while running HyperCard on an Macintosh with 2 megabytes of RAM, could not be opened on a Macintosh with 1 megabyte. The current useful size of a card (or background) is therefore between 50 and 100 kilobytes.

The term *part,* in this appendix and internally in HyperCard, refers to buttons or fields. The value represented by `LongInt` is 2,147,483,647; the value represented by `Integer` is 32,767.

■ **Table G-1** HyperCard limits

Item	Limit
Stack limits	
Stack size	512 megabytes
Minimum stack size	4896 bytes
Maximum total number of bitmaps, cards, and backgrounds per stack	16,777,216
Maximum stack name size	31 characters
Maximum stack script size	30,000 characters
Background limits	
Background size (bytes)	`LongInt`*
Minimum background size	64 bytes
Maximum parts per background	`Integer`
Maximum total part size per background (bytes)	`LongInt`
Maximum background name size	31 characters
Maximum background script size	30,000 characters
Card limits	
Card size (bytes)	`LongInt`*
Minimum card size	64 bytes
Maximum parts per card	`Integer`
Maximum total part size per card (bytes)	`LongInt`
Maximum total text size per card (bytes)	`LongInt`
Maximum card name size	31 characters
Maximum card script size	30,000 characters
Part (button or field) limits	
Part size (bytes)	`Integer`**
Minimum overhead per part	30 bytes
Maximum part name size	31 characters
Maximum part text size	30,000 characters
Maximum part script size	30,000 characters

[continued]

■ **Table G-1** HyperCard limits [continued]

Item	Limit
HyperTalk limits	
Maximum nested `repeat` structures	32
Maximum active variables (all pending handlers)	512
Maximum size card name with `go` command	31 characters
Maximum variable name size	31 characters
Maximum number format size	31 characters
Maximum size of command with arguments	254 characters
Maximum handler name size	254 characters
Maximum file I/O buffer size	16,384 bytes
Maximum script size	30,000 characters
Maximum variable value size	Limited by available memory

* Limited by HyperCard stack size; less than 100 kilobytes for practical use.

** The sum of the other elements in the button or field must be less than the part size.

Appendix H **HyperCard Syntax Summary**

This appendix lists the HyperTalk built-in commands and functions, showing the syntax of their parameters. It also provides a table that defines the metasymbols used in the syntax for commands, functions, and properties. The metasymbol table includes example expressions and terms that you can use as a guide when you substitute actual instances for the metasymbols in your scripts.

HyperTalk built-in commands and functions are described in more detail in Chapters 10 and 11, respectively. A brief description is included in Appendix I.

Syntax description notation

The syntax descriptions use the following typographic conventions. Words or phrases in `this type` are HyperTalk language elements that you type to the computer literally, exactly as shown. Words in *italic* type are metasymbols (used to describe general elements) not specific names—you must substitute the actual instances. Square brackets ([ ]) enclose optional elements that may be included if you need them. (Don't type the square brackets.) In some cases, optional elements change what the message does; in other cases they are helper words that have no effect except to make the message more readable.

It doesn't matter whether you use uppercase or lowercase letters; names that are formed from two words are shown in lowercase letters with a capital in the middle (`likeThis`) merely to make them more readable. The HyperTalk prepositions `of` and `in` are interchangeable—the syntax descriptions use the one that sounds more natural.

The terms *factor* and *expression* are defined in Chapter 7. Briefly, a factor can be a constant, literal, function, property, number, or container, and an expression can be a factor or a complex expression built with factors and operators. Also, a factor can be an expression within parentheses.

■ **Table H-1** HyperTalk command syntax

Commands
`add` *number* `to [`*chunk* `of]` *container*
`answer` *question* `with` *reply*
`answer` *question* `with` *reply1* `or` *reply2*
`answer` *question* `with` *reply1* `or` *reply2* `or` *reply3*
`answer file [`*promptText*`] [of type` *fileType*`]`
`arrowKey` *direction*
`ask` *question* `[with` *defaultAnswer*`]`
`ask password [clear]` *question* `[with` *defaultAnswer*`]`
`ask file` *promptText* `[with [default]` *fileName*`]`
`beep [`*number*`]`
`choose` *toolName* `tool`
`choose tool` *toolNumber*

[continued]

■ **Table H-1** HyperTalk command syntax [continued]

Commands
`click at` *point* `[with` *key*`]`
`close file` *fileName*
`close printing`
`controlKey` *keyNumber*
`convert [`*chunk* `of]` *container* `to` *format* `[and` *format*`]`
`create menu` *menuName*
`create stack` *stackName* `[with` *background*`] [in a new window]`
`debug checkpoint`
`delete` *chunk* `[of` *container*`]`
`delete` *menu*
`delete` *menuItem* `of` *menu*
`dial` *number*
`dial` *number* `with modem [`*modemCommands*`]`
`disable` *menu*
`disable` *menuItem* `of` *menu*
`divide [`*chunk* `of]` *container* `by` *number*
`doMenu` *itemName* `[without dialog]`
`drag from` *point1* `to` *point2*
`drag from` *point1* `to` *point2* `with` *key*
`edit script of` *object*
`enable` *menu*
`enable` *menuItem* `of` *menu*
`enterKey`
`find` *text* `[in` *field*`]`
`find chars` *text* `[in` *field*`]`
`find string` *text* `[in` *field*`]`
`find whole` *text* `[in` *field*`]`
`find word` *text* `[in` *field*`]`
`functionKey` *keyNumber*
`get` *expression*
`go [to] [stack]` *stackName*
`go [to]` *background* `[of [stack]` *stackName*`]`

[continued]

■ **Table H-1** HyperTalk command syntax [continued]

Commands

```
go [to] card [of background] [of [stack] stackName]¬
[in a new window] [without dialog]
help
hide menuBar
hide object
hide card picture
hide background picture
hide picture of background
hide titleBar
hide card window
hide window windowName
lock screen
mark card
mark cards where expression
mark all cards
mark cards by finding text [in field]
mark cards by finding chars text [in field]
mark cards by finding string text [in field]
mark cards by finding whole text [in field]
mark cards by finding word text [in field]
multiply [chunk of] container by number
open [fileName with] application
open file fileName
open printing [with template templateName] [with dialog]
open report printing [with dialog]
open report printing [with template templateName]
palette paletteName[, point]
picture [fileName, fileType, windowStyle, visible, depth]
play sound [tempo] [notes]
play stop
pop card [preposition [chunk of] container]
print card [from point1 to point2]
print expression
print fileName with application
print field
print marked cards
print number cards
```

[continued]

■ **Table H-1** HyperTalk command syntax [continued]

Commands
`push` *card*
`put` *expression*
`put` *expression* `into [`*chunk* `of]` *container*
`put` *expression* `after [`*chunk* `of]` *container*
`put` *expression* `before [`*chunk* `of]` *container*
`put` *itemName preposition* `[`*menuItem* `of]` *menu* `[with menuMsg` *message*`]`
`read from file` *fileName* `until` *char*
`read from file` *fileName* `at` *start* `for` *numberOfChars*
`read from file` *fileName* `for` *numberOfChars*
`reset menuBar`
`reset paint`
`returnKey`
`save stack` *stackName* `as` *fileName*
`select` *object*
`select [`*preposition*`]` *chunk* `of` *field*
`select [`*preposition*`] text of` *field*
`select empty`
`set [the]` *property* `[of` *element*`] to` *value*
`show groups`
`show` *object* `[at` *point*`]`
`show marked cards`
`show` *number* `cards`
`show menuBar`
`start using stack` *stackName*
`stop using stack` *stackName*
`subtract` *number* `from [`*chunk* `of]` *container*
`tabKey`
`type` *text*
`type` *text* `with` *key*
`type` *text* `with` *key1*, *key2*
`type` *text* `with` *key1*, *key2*, *key3*
`unlock screen`
`unmark` *card*
`unmark cards where` *expression*
`unmark all cards`
`unmark cards by finding` *text* `in` *field*
`unmark cards by finding chars` *text* `in` *field*

[continued]

■ **Table H-1** HyperTalk command syntax [continued]

Commands
`unmark cards by finding string` *text* `in` *field*
`unmark cards by finding whole` *text* `in` *field*
`unmark cards by finding word` *text* `in` *field*
`visual [effect]` *effectName* `[`*speed*`]`
`visual [effect]` *effectName* `[`*speed*`] to` *image*
`wait [for]` *time* `[seconds]`
`wait until` *condition*
`wait while` *condition*
`write` *text* `to file` *fileName*
`write` *text* `to file` *fileName* `at` *start*

■ **Table H-2** HyperTalk function syntax

Functions
`the abs of` *factor*
`abs(`*expression*`)`
`annuity(`*rate*`,` *periods*`)`
`the atan of` *factor*
`atan(`*expression*`)`
`average(`*list*`)`
`the clickChunk`
`the charToNum of` *factor*
`charToNum(`*expression*`)`
`clickChunk()`
`the clickLine`
`clickLine()`
`the clickLoc`
`clickLoc()`
`the clickText`
`clickText()`
`the commandKey`
`commandKey()`
`compound(`*rate*`,` *periods*`)`

[continued]

■ **Table H-2** HyperTalk function syntax [continued]

Functions
`the cos of` *factor*
`cos(`*expression*`)`
`the [`*adjective*`] date`
`the diskSpace`
`diskSpace()`
`the exp of` *factor*
`exp(`*expression*`)`
`the exp1 of` *factor*
`exp1(`*expression*`)`
`the exp2 of` *factor*
`exp2(`*expression*`)`
`the foundChunk`
`foundChunk()`
`the foundField`
`foundField()`
`the foundLine`
`foundLine()`
`the foundText`
`foundText()`
`the length of` *factor*
`length(`*expression*`)`
`the ln of` *factor*
`ln(`*expression*`)`
`the ln1 of` *factor*
`ln1(`*expression*`)`
`the log2 of` *factor*
`log2(`*expression*`)`
`max(`*list*`)`
`the menus`
`menus()`
`min(`*list*`)`

[continued]

■ **Table H-2** HyperTalk function syntax [continued]

Functions
`the mouse`
`mouse()`
`the mouseClick`
`mouseClick()`
`the mouseH`
`mouseH()`
`the mouseLoc`
`mouseLoc()`
`the mouseV`
`mouseV()`
`[the] number of` *objects*
`[the] number of` *chunks* `in` *expression*
`[the] number of backgrounds [in this stack]`
`[the] number of cards in` *background*
`[the] number of cards [in this stack]`
`[the] number of marked cards`
`[the] number of menus`
`[the] number of menuItems of` *menu*
`[the] number of windows`
`the numToChar of` *factor*
`numToChar(`*expression*`)`
`offset(`*string1*`,` *string2*`)`
`the optionKey`
`optionKey()`
`the param of` *factor*
`param(`*expression*`)`
`the paramCount`
`paramCount()`
`the params`
`params()`
`the random of` *factor*
`random(`*expression*`)`
`the result`
`result()`
`the round of` *factor*
`round(`*expression*`)`

[continued]

■ **Table H-2** HyperTalk function syntax [continued]

Functions
`the screenRect`
`screenRect()`
`the seconds`
`seconds()`
`the selectedChunk`
`selectedChunk()`
`the selectedField`
`selectedField()`
`the selectedLine`
`selectedLine()`
`the selectedText`
`selectedText()`
`the shiftKey`
`shiftKey()`
`the sin of` *factor*
`sin(`*expression*`)`
`the sound`
`sound()`
`the sqrt of` *factor*
`sqrt(`*expression*`)`
`the tan of` *factor*
`tan(`*expression*`)`
`the target`
`target()`
`the ticks`
`ticks()`
`the [`*adjective*`] time`
`time()`
`the tool`
`tool()`
`the trunc of` *factor*
`trunc(`*expression*`)`
`the value of` *factor*
`value(`*expression*`)`
`the [long] version [of HyperCard]`
`version()`
`the version of stack` *stackName*
`the windows`
`windows()`

■ **Table H-3** HyperTalk metasymbol definitions

Metasymbol	Definition
adjective	One of the following: `abbreviated`, `abbrev`, `abbr`, `long`, `short`
alignment	One of the following: `left`, `center`, `right`
application	A valid Macintosh application name. The name must include a valid pathname for that application. For example: `"HD20:Applications:WonderArt"`
background	An expression that identifies a background by name, number, or ID. For example: `background 1`, `bkgnd id 2488`, `bg "Fish"`, `this bg`, `next bg`, `previous bg`, `prev bg`, `first bg`, `middle bg`, `last bg`, `the target`, `me`
boolean	An expression that yields either `true` or `false`
button	An expression that identifies a button by name, number, or ID. For example: `card button 1`, `card btn id 4322`, `background button "Left"`, `the target`, `me`
card	An expression that identifies a card by name, number, or ID. For example: `card 1`, `card id 1234`, `cd "Birds"`, `this card`, `next card`, `first card`, `previous card`, `prev card`, `mid card`, `last card`, `recent card`, `the target`, `me`
char	A chunk name representing a character in a source (spaces count as characters).
chunk	The words `character`, `char`, `word`, `item`, or `line` preceded by an ordinal or followed by an integer, range of integers, or another chunk expression. For example: `first character`, `second char`, `middle word`, `third item`, `char 1 to 6`, `word 2 to 4`, `third word`, `item 3`, `word 3 of line 1 in card field 2 of card "Eagle"`
chunks	The plural form of a chunk. For example: `characters`, `chars`, `words`, `items`, `lines`

[continued]

■ **Table H-3** HyperTalk metasymbol definitions [continued]

Metasymbol	Definition
condition	Evaluates to `true` or `false` (used with the `wait while` and `wait until` forms of the `wait` command). For example: `the mouse is down`
container	Evaluates to a variable, the selection, the Message box, or any expression that identifies a field by name, number, or ID. For example: `myVar, the selection, the message box, msg box, card field 3, bg fld "Winners"`
cursorType	An expression that yields the name or number of one of the HyperCard cursors. Only four cursors have number equivalents. For example: `Ibeam (1), cross (2), plus (3), watch (4), hand, arrow, busy, none`
debuggerName	An expression that yields the name of a debugger XCMD
defaultAnswer	An expression that yields a text string used as the default text in the `ask` command dialog box
designator	An expression that yields a name or number of an icon resource
depth	An expression that yields a number representing the bit depth to use for the `picture` command off-screen buffer (values from 1 through 32 are supported)
direction	An expression that yields one of the arrow keys: `left`, `right`, `up`, or `down`
effectName	Evaluates to one of the following: `barn door close, barn door open, checkerboard, dissolve (fade), iris close, iris open, plain, scroll down, scroll left, scroll right, scroll up, venetian blinds, wipe down, wipe left, wipe right, wipe up, zoom close, zoom in, zoom open, zoom out`
expression	Evaluates to a constant, number, or text. For example: `true, 21-18, "My shoes are" & It, avg(39,45,67,79), 4.25, (the number of lines in field 2)`

[continued]

■ **Table H-3** HyperTalk metasymbol definitions [continued]

Metasymbol	Definition
factor	A single element of value in an expression. For example: `(24*4+4)`, `(the number of cards in this stack)`, `field 2`, `line 1`, `.23`, `7`, `-4`, `("text")`
field	An expression that identifies a field by name, number, or ID. For example: `card field 1`, `fld id 4322`, `background field "Left"`, `the target`, `me`
fileName	An expression that yields a valid Macintosh filename. The filename could be the name of a stack or a file belonging to another application.
fileType	An expression that yields a valid Macintosh file type. The file types are `stack`, `text`, `application`, `paint`, `resource`, `file`, `clipboard`
font	An expression that yields the name of one of the fonts in the system. For example: `Geneva`, `Palatino`, `New York`, `Monaco`
format	One of the following: `seconds`, `dateItems`, `long date`, `short date`, `abbreviated date`, `long time`, `short time`, `abbreviated time`
formatType	An expression that yields the format (within quotation marks) that is to be used for the display of numbers. For example: `0.`, `00.##`, `0`
image	One of the following: `black`, `card`, `gray`, `grey`, `inverse`, `white`, `color` *integer* (*Integer* is an expression that yields a positive integer in the range 1 through 16.)
itemName	An expression that yields a menu item, surrounded by quotation marks, in one of the HyperCard menus. For example: `"New Stack..."`, `"New Card..."`, `"Paste"`, `"Copy"`, `"Card Info..."`
key	One of the following: `shiftKey`, `optionKey`, `commandKey`, `cmdKey`

[continued]

■ **Table H-3** HyperTalk metasymbol definitions [continued]

Metasymbol	Definition
keyNumber	An expression that yields a positive integer in the range 1 through 126. The integer represents one of the keys on the keyboard. See Table 10-1 for definitions of the *keyNumber* values.
languageName	An expression that yields the name of a language. For example: `French, Swedish, English`
list	A comma-separated list of numbers. For example: `(1,3,5,79)`
location	A point: two integers separated by a comma, representing horizontal and vertical pixel offsets (respectively) from the upper-left corner of the card window. For example: `"94,109", "94" "&" "," "&" "109", the clickLoc`
menu	An expression that yields the name or number of a HyperCard menu or a custom menu. For example: `menu "Objects", menu "Apple", menu "Home", menu 4, sixth menu`
menuItem	An expression that yields the name or number of a HyperCard menu item or a custom menu item. For example: `menuItem "New Stack...", menuItem "New Card...", menuItem "Pictures", menuItem 3, fifth menuItem`
menuNumber	An expression that yields a menu descriptor in menu form. For example: `menu4, sixth menu`
message	An expression that yields the message to send when a menu item is chosen. For example: `"Go next card", "Go stack Birds"`
messageWindow	An expression that yields the name of a message watcher XCMD
modemCommands	A text expression that is a valid command for your modem
notes	An expression that yields a list of one or more notes representing the pitch at which the sound plays and the duration of the notes
number	An expression that yields a number. For example: `2, 55, 4.5, the number of cards - 4`
numberOfChars	An expression that yields a positive integer. For example: `2, 45, 6, (the number of chars in line 2)`

[continued]

■ **Table H-3** HyperTalk metasymbol definitions [continued]

Metasymbol	Definition
object	An expression that identifies a background, button, card, field, stack, or window. For example: `tool window`, `message box`, `first bg`, `button 1`, `the next card`, `pattern window`, `card field id 1456`, `stack "Puzzle"`
objectName	An expression that yields any valid HyperCard object name (the object name can be a maximum of 31 characters)
objects	An expression that yields a group of objects: For example: `backgrounds`, `cards`, `marked cards`, `cards in background 4`, `windows`, `menus`, `menuItems in menu "Go"`, `words in field 3`
objectStyle	An expression that yields one of the valid field or button styles. Button styles are `transparent`, `opaque`, `rectangle`, `roundRect`, `checkBox`, or `radioButton`. Field styles are `transparent`, `opaque`, `rectangle`, `shadow`, or `scrolling`.
periods	An expression that yields a number that represents the number of periods over which a value is calculated
point	An expression that yields a point, reported as two integers separated by a comma. For example, `100,100`, `45,150`, `75,135`
preposition	One of the following: `into`, `before`, `after`
promptText	Evaluates to a text string surrounded by quotation marks. The text prompts the user for an action. For example: `"Select a Paint file:"`, `"Save this stack as:"`, `"Select a text file:"`
property	An expression that yields one of the HyperCard properties.
question	An expression that yields a text string. The text prompts the user to make a selection. For example: `"Where's Dave?"`, `"Are you sure you want to end this session?"`, `"Do you want to quit or open another stack?"`

[continued]

■ **Table H-3** HyperTalk metasymbol definitions [continued]

Metasymbol	Definition
rate	An expression that yields a number that represents the interest rate per period (see *periods*)
rectangle	An expression that yields four positive integers separated by commas. The integers are the left, top, right, and bottom, respectively, of a rectangle in pixels. For example: `78,78,340,440`
reply	An expression that yields a text string. The text is a choice used for a response to a question. For example: `"OK"`, `"Cancel"`, `"Do it!"`, `"Trash it!"`
scriptText	An expression that yields any valid container that contains a script or yields a text string that is a script (see *container*)
scrollingField	Any valid card or background scrolling field
sortDirection	One of the following: `ascending`, `descending`
sortStyle	One of the following: `text`, `numeric`, `dateTime`, `international`
sound	An expression that yields the name of a digitized sound. For example, the sounds `boing` and `harpsichord` are included with HyperCard.
speed	One of the following: `fast`, `very fast`, `slow`, `slowly`, `very slow`, `very slowly`
stack	An expression that yields a valid stack descriptor preceded by the word *stack*. For example: `stack "HyperCard:Stacks:MyStack"`
stackName	An expression that yields a valid stack name. For example: `"HyperCard:Stacks:MyStack"`
start	An expression that yields a positive integer that indicates the starting point (offset from the beginning of the file) from which reading or writing occurs

[continued]

■ **Table H-3** HyperTalk metasymbol definitions [continued]

Metasymbol	Definition
string	An expression that yields a text string. For example: `"The rabbit"`, `word 2 in field 3 & space & word 3 in field 3`, `"My feet" && It`
style	An expression that yields a value of `plain` or any combination of the following: `bold`, `italic`, `underline`, `outline`, `shadow`, `condensed`, `extend`, and `group` (separated by commas)
templateName	An expression that yields the name of a print template
tempo	A number specifying the speed at which the group of notes is played. (120 is a medium tempo for HyperCard; higher numbers are faster.) For example: `tempo 200`, `tempo 120`
text	An expression that yields a one or more text strings
time	An expression that yields a positive integer. For example: `10`, `60`
toolName	One of the following: `browse`, `brush`, `bucket`, `button`, `curve`, `eraser`, `field`, `lasso`, `line`, `oval`, `pencil`, `poly[gon]`, `rect[angle]`, `reg[ular] poly[gon]`, `round rect[angle]`, `select`, `spray [can]`, `text`
toolNumber	One of the numbers (`1` through `18`) that correspond to the tools in the Tools palette
value	An expression that yields a valid setting for the specified property
variableWatcher	An an expression that yields the name of a variable watcher window
visualEffect	An expression that yields any of the forms of the `visual` command. For example: `scroll down`, `scroll left`, `wipe up`, `fade`
windowStyle	An expression that yields one of the window styles supported by the `picture` command. For example: `plain`, `rect`, `zoom`, `roundRect`, `dialog`, `document`, `shadow`, and `windoid`

Appendix I **HyperTalk Vocabulary**

This appendix lists and defines, in alphabetical order, the HyperTalk native vocabulary—the names of its built-in commands and functions, its system messages, keywords, the names of objects and their properties, and various adjectives, constants, ordinals, and other terms.

This list is not exhaustive—there are other terms with specific meanings recognized by HyperCard in particular contexts, and they are described with the primary term to which they relate. For example, the names of the various visual effects are listed with the `visual` command in Chapter 10.

Appendix H shows HyperTalk built in commands and functions parameter syntax.

■ **Table I-1** HyperTalk vocabulary

Term	Category	Meaning
`abbr[ev[iated]]`	Adjective	Modifies the value returned by the `date` function or the `name` or `ID` property.
`abs`	Function	Returns the absolute value of a number.
`add`	Command	Adds the value of an expression to a value in a container.
`after`	Preposition	Used with the `put` command, directing HyperCard to append a new value following any preexisting value in a container.
`all`	Adjective	Specifies total number of cards in stack to `show cards` command.
`annuity`	Function	Computes present or future value of an ordinary annuity.
`answer`	Command	Displays a dialog box with question and reply buttons.
`answer file`	Command	Presents the standard dialog box for locating a file; used for opening files of a specified type.
`any`	Ordinal	Special ordinal used with object or chunk to specify a random element within its enclosing set.
`arrowKey`	Command	Takes you to another card.
`arrowKey`	System message	Sent to current card when an arrow key is pressed.
`ask`	Command	Displays a dialog box with a question and default answer.
`ask file`	Command	Presents the standard dialog box for locating where to save a file; used for saving files.
`ask password`	Command	Displays a dialog box with a field for a password.
`atan`	Function	Returns the trigonometric arc tangent of a number.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`autoHilite`	Property	Determines whether or not the specified button's `hilite` property is affected by the message `mouseDown`.
`autoTab`	Property	Determines whether the specified nonscrolling field sends the `tabKey` message to the current card.
`average`	Function	Returns the average value of numbers in a list.
`background`	Object	Generic name of background object; used with specific designation (`go to next background`). Also used to specify containing object for buttons and, optionally, fields (`background button 2`).
`backgrounds`	Object type	Specifies backgrounds as type of object to the `number` function.
`beep`	Command	Causes the Macintosh to make a beep sound.
`before`	Preposition	Used with `put` command, directing HyperCard to place a new value at the beginning of any preexisting value in a container.
`bkgnd`	Object	Abbreviation for `background`.
`bkgnds`	Object type	Specifies backgrounds as type of object to the `number` function.
`bg`	Object	Abbreviation for `background`.
`blindTyping`	Property	Allows typing into the Message box when it's hidden.
`bottom`	Property	Determines or changes the value of item 4 of the `rectangle` property when applied to the specified object or window.
`bottomRight`	Property	Determines or changes items 3 and 4 of the value of the `rectangle` property when applied to the specified object or window.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`browse`	Tool	Name of tool from Tools palette; used with the `choose` command or returned by the `tool` function.
`brush`	Property	Determines the current brush shape.
`brush`	Tool	Name of tool from Tools palette; used with the `choose` command or returned by the `tool` function.
`btn`	Object	Abbreviation for `button`.
`bucket`	Tool	Name of tool from Tools palette; used with the `choose` command or returned by the `tool` function.
`button`	Object	Generic name of button object; used with a specific designation (`hide button one`).
`button`	Tool	Name of tool from Tools palette; used with `choose` command or returned by the `tool` function.
`buttonCount`	Property	Determines the number of buttons in an open palette XCMD.
`buttons`	Object type	Specifies buttons as type of object to the `number` function.
`cantAbort`	Property	Determines if a script can be stopped by pressing Command-period.
`cantDelete`	Property	Determines if a background, card, or stack can be deleted while the user level is Browsing.
`cantModify`	Property	Determines if a stack can be modified while the user level is Browsing. Can be used with a password to prevent anyone without the password from modifying a stack.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`cantPeek`	Property	Determines if the outline is shown around buttons and fields when the Command-Option or Command-Option-Shift keys are pressed.
`card`	Object	Generic name of a card object; used with a specific designation (`go to card "fred"`). Also used to specify containing object for fields and, optionally, buttons (`card field "date"`).
`cards`	Object type	Specifies cards as type of object to the `number` function.
`cd`	Object	Abbreviation for `card`.
`center`	Adjective	Specifies center alignment of text in a field.
`centered`	Property	Determines whether shapes are drawn from the center or from the corner.
`char[acter]`	Chunk	A character of text in any container or expression.
`char[acter]s`	Chunk type	Specifies characters as type of chunk to the `number` function.
`charToNum`	Function	Returns ASCII value of a character.
`checkMark`	Property	Determines check character for a menu item.
`choose`	Command	Changes the current tool.
`click`	Command	Causes same actions as clicking at a specified location.
`clickChunk`	Function	Returns chunk information about text that is clicked.
`clickLine`	Function	Returns line information about text that is clicked.
`clickLoc`	Function	Returns location of most recent click.
`clickText`	Function	Returns text information about word or group phrase that is clicked.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`closeBackground`	System message	Sent to current card just before you leave the current background.
`closeCard`	System message	Sent to current card just before you leave it.
`closeField`	System message	Sent to unlocked field when it closes.
`closeStack`	System message	Sent to current card just before you leave the current stack.
`close file`	Command	Closes a previously opened disk file.
`close printing`	Command	Ends a print job.
`cmdChar`	Property	Determines the character to use with the Command key to invoke a menu item.
`commandKey`	Function	Sent to current card when a combination of the Command key and another key is pressed.
`commandKeyDown`	System message	Returns the Command key.
`commands`	Property	Returns a list of the commands associated with the buttons in a palette XCMD.
`compound`	Function	Computes present or future value of a compound interest–bearing account.
`controlKey`	Command	Sends the `controlKey` system message.
`controlKey`	System message	Sent to current card when a combination of the Control key and another key is pressed.
`convert`	Command	Converts a date or time to specified format.
`create menu`	Command	Creates a new menu with the specified name.
`create stack`	Command	Creates a new stack with the specified name and background.
`cos`	Function	Returns the cosine of the angle that is passed to it.
`cursor`	Property	Sets image appearing at pointer location on screen. You can only set `cursor`; you can't get it.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`curve`	Tool	Name of tool from Tools palette; used with the `choose` command or returned by the `tool` function.
`date`	Function	Returns a string representing the current date.
`debug checkpoint`	Command	Sets a checkpoint in a script to invoke the built-in debugger.
`debugger`	Property	Determines the debugger to use.
`delete` (text)	Command	Removes a chunk of text from a container.
`delete` (menu)	Command	Deletes a menu without a dialog box.
`delete` (menu items)	Command	Deletes a menu item without a dialog box.
`deleteBackground`	System message	Sent to the current card just before the background is deleted.
`deleteButton`	System message	Sent to a button just before it is deleted.
`deleteCard`	System message	Sent to current card just before it is deleted.
`deleteField`	System message	Sent to a field just before it is deleted.
`deleteStack`	System message	Sent to the current card just before a stack is deleted.
`dial`	Command	Generates touch-tone sounds through audio output or a modem attached to serial port.
`disable`	Command	Disables the specified menu or menu item.
`diskSpace`	Function	Displays the amount of free space available on the disk containing the current stack.
`dithering`	Property	Determines whether or not the picture opened by the `picture` command is dithered.
`divide`	Command	Divides the value in a container by the value of an expression.
`do`	Command	Sends the value of an expression as a message to the current card.
`do`	Keyword	Sends the value of an expression as a message to the current card.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`doMenu`	Command	Performs a specified menu command.
`doMenu`	System message	Sent to the current card when any menu item is chosen.
`dontSearch`	Property	Determines whether a card, background, or background field can be searched by the `find` command.
`dontWrap`	Property	Determines whether the text in a field wraps onto the next line.
`down`	Constant	Value returned by various functions to describe the state of a key or the mouse button.
`drag`	Command	Performs same action as a manual drag.
`dragSpeed`	Property	Sets pixels-per-second speed at which the pointer moves with the `drag` command.
`editBkgnd`	Property	Determines whether manipulation of buttons, fields, or paintings occurs on the current card or background.
`edit script`	Command	Opens the script of a specified object.
`eight`	Constant	String representation of the numerical value 8.
`eighth`	Ordinal	Designates object or chunk number eight within its enclosing set.
`else`	Keyword	Optionally follows `then` clause in an `if` structure to introduce an alternative action clause.
`empty`	Constant	The null string; same as the literal `""`.
`enable`	Command	Enables the specified menu or menu item.
`enabled`	Command	Determines whether the specified menu or menu item is enabled.
`end`	Keyword	Marks the end of a message handler, function handler, `repeat` loop, or multiple-statement `then` or `else` clause of an `if` structure.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`enterInField`	System message	Sent to the field when the Enter key is pressed while there is an insertion point or selection in the field.
`enterKey`	Command	Sends contents of the Message box to the current card.
`enterKey`	System message	Sent to the current card when the Enter key is pressed unless the text insertion point is in a field.
`eraser`	Tool	Name of tool from Tools palette; used with `choose` command or returned by the `tool` function.
`exit`	Keyword	Immediately ends execution of a message handler, function handler, or `repeat` loop.
`exp`	Function	Returns the mathematical exponential of its argument.
`exp1`	Function	Returns one less than the mathematical exponential of its argument.
`exp2`	Function	Returns the value of 2 raised to the power specified by the argument.
`export paint`	Command	Creates a Macintosh paint file with the image of the current card.
`false`	Constant	Boolean value resulting from evaluation of a comparative expression and returned from some functions.
`field`	Container	Generic name of field container; used with specific designation (`put the time into card field "time"`).
`field`	Object	Generic name of field object; used with specific designation (`get name of first field`).

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`field`	Tool	Name of tool from Tools palette; used with the `choose` command or returned by the `tool` function.
`fields`	Object type	Specifies fields as type of object to the `number` function.
`fifth`	Ordinal	Designates object or chunk number five within its enclosing set.
`filled`	Property	Determines the Draw Filled setting.
`find`	Command	Searches card and background fields for text strings derived from an expression.
`first`	Ordinal	Designates object or chunk number one within its enclosing set.
`five`	Constant	String representation of the numerical value 5.
`fixedLineHeight`	Property	Determines whether or not a field has fixed line spacing.
`formFeed`	Constant	The form feed character (ASCII 12), which starts a new page in some file formats.
`foundChunk`	Function	Returns a chunk expression describing the text found with the `find` command.
`foundField`	Function	Returns an expression describing the field the text was found in with the `find` command.
`foundLine`	Function	Returns an expression describing the line the text was found in with the `find` command.
`foundText`	Function	Returns the text found with the `find` command.
`four`	Constant	String representation of the numerical value 4.
`fourth`	Ordinal	Designates object or chunk number four within its enclosing set.
`freeSize`	Property	Determines the amount of free space available in a specified stack.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`function`	Keyword	Marks the beginning of a function handler. Connects the handler with a particular function call.
`functionKey`	Command	Performs Undo, Cut, Copy, or Paste operations with parameter values of 1, 2, 3, or 4, respectively.
`functionKey`	System message	Sent to current card when any function key on the Apple Extended Keyboard is pressed.
`get`	Command	Puts the value of an expression into the local variable `It`.
`global`	Keyword	Declares specified variables to be valid beyond current execution of current handler.
`globalLoc`	Property	Determines the location of a window created with the `picture` command in global coordinates.
`globalRect`	Property	Determines the rectangle of a window created with the `picture` command in global coordinates.
`go`	Command	Takes you to a specified card or stack.
`grid`	Property	Determines the Grid setting.
`hBarLoc`	Property	Determines the location of the horizontal bar in the Variable Watcher window.
`help`	Command	Takes you to the first card in the stack named Help.
`height`	Property	Determines or changes the vertical distance in pixels occupied by the rectangle of the specified button or field.
`hide`	Command	Hides the specified window from view.
`hide groups`	Command	Hides the gray underline displayed beneath text by the `show groups` command.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`hideIdle`	Property	Determines whether or not the "Hide idle" check box is checked in the Message Watcher window.
`hide menuBar`	Command	Hides the HyperCard menu bar.
`hideUnused`	Property	Determines whether or not the "Hide unused messages" check box is checked in the Message Watcher window.
`hilite`	Property	Determines whether a specified button is highlighted.
`hilitedButton`	Property	Determines whether a button in a palette XCMD is highlighted.
`icon`	Property	Determines the icon that is displayed with a specified button.
`ID`	Property	Determines the permanent ID number of a specified background, card, field, or button.
`idle`	System message	Sent to the current card repeatedly whenever nothing else is happening.
`if`	Keyword	Introduces a conditional structure containing statements to be executed only if a specified condition is true.
`import paint`	Command	Reads in a Macintosh paint file and makes it the current selection.
`in`	Operator	Used with the comparison operators `is in` and `is not in`.
`in`	Preposition	Used as a connective preposition in chunk expressions. For example, `card 12 in this stack`
`into`	Preposition	Used with the `put` command, directing HyperCard to replace any preexisting value in a container with a new value.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`It`	Container	Local variable that is the default destination for the `get`, `ask`, `answer`, `read`, and `convert` commands.
`item`	Chunk	A piece of text delimited by commas in any container or expression.
`items`	Chunk type	Specifies items as type of chunk to the `number` function.
`keyDown`	System message	Sent to the current card when a key is pressed.
`language`	Property	Used to choose language in which scripts are displayed.
`lasso`	Tool	Name of tool from Tools palette; used with the `choose` command or returned by the `tool` function.
`last`	Ordinal	Special ordinal used with object or chunk to specify the element whose number is equal to the total number of elements in its enclosing set.
`length`	Function	Returns the number of characters in the text string derived from an expression.
`left`	Adjective	Specifies left-justified alignment of text in a field.
`left`	Property	Determines or changes the value of item 1 of the `rectangle` property when applied to the specified object or window.
`line`	Chunk	A piece of text delimited by return characters in any container.
`line`	Tool	Name of tool from Tools palette; used with the `choose` command or returned by the `tool` function.
`lineFeed`	Constant	The line feed character (ASCII 10), which starts a new line in some file formats.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`lines`	Chunk type	Specifies lines as type of chunk to the `number` function.
`lineSize`	Property	Determines the thickness of lines drawn with line and shape tools.
`ln`	Function	Returns the base-*e* (natural logarithm) of the number passed to it.
`ln1`	Function	Returns the base-*e* (natural logarithm) of the sum of the number passed to it plus 1.
`loc`	Property	Determines the location at which a picture window created with the `picture` command is displayed.
`loc[ation]`	Property	Determines the location at which a window, field, or button is displayed.
`lockMessages`	Property	Allows or prevents HyperCard from sending all automatic messages such as `openCard`.
`lockRecent`	Property	Allows or prevents HyperCard from adding miniature representations to the Recent card.
`lock screen`	Command	Prevents updating of the screen from card to card.
`lockScreen`	Property	Determines whether the screen is updated when moving from card to card.
`lockText`	Property	Determines whether text editing is allowed in a specified field.
`log2`	Function	Returns the base-2 logarithm of the number passed to it.
`long`	Adjective	Modifies value returned by `date` function and by `name` and `ID` properties.
`longWindowTitles`	Property	Determines whether the window title bar contains the full pathname of a stack or the short name.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`mark`	Command	Marks cards.
`markChar`	Property	Determines the check mark character used to indicate a menu item is choosen.
`marked`	Property	Determines whether or not a specified card is marked.
`max`	Function	Returns the highest-value number from a list of numbers.
`me`	Object	The object containing the executing handler.
`menuMsg`	Property	Determines the message to be sent by a specified menu item.
`menu`	Function	Returns a list of the menu items in a specified menu.
`menus`	Function	Returns a list of the menu names in the HyperCard menu bar.
`message [box]`	Container	The Message box.
`messageWatcher`	Property	Determines the message watcher to use.
`mid[dle]`	Ordinal	Special ordinal used with an object or chunk to specify the element whose number is equal to one more than half the total number of elements in its enclosing set.
`min`	Function	Returns the lowest-value number from a list of numbers.
`mouse`	Function	Returns state of the mouse button: `up` or `down`.
`mouseClick`	Function	Returns whether the mouse button has been clicked.
`mouseDown`	System message	Sent to a button, unlocked field, or the current card when the mouse button is pressed down.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`mouseDownInPicture`	System message	Sent to the current card when the mouse is down in a window created with the `picture` command.
`mouseEnter`	System message	Sent to a button or field when the pointer is first moved inside its rectangle.
`mouseH`	Function	Returns the horizontal offset in pixels of the pointer from the left edge of the card window.
`mouseLeave`	System message	Sent to a button or field when the pointer is first removed from its rectangle.
`mouseLoc`	Function	Returns the point on the screen where the pointer is currently located.
`mouseStillDown`	System message	Sent to a button, unlocked field, or the current card repeatedly when the mouse button is held down.
`mouseUp`	System message	Sent to a button, unlocked field, or the current card when the mouse button is released after having been previously pressed down within the same object's rectangle.
`mouseUpInPicture`	System message	Sent to the current card when the mouse is released after being down in a window created with the `picture` command.
`mouseV`	Function	Returns the vertical offset in pixels of the pointer from the top of the screen.
`mouseWithin`	System message	Sent to a button or field repeatedly while the pointer remains inside its rectangle.
`msg [box]`	Container	The Message box.
`multiple`	Property	Determines whether multiple images are drawn with a shape tool.
`multiply`	Command	Multiplies the value in a container by the value derived from an expression.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`multiSpace`	Property	Determines the space between objects drawn when the `multiple` property is `true`.
`name`	Property	Determines the name of a stack, background, card, field, button, menu, or menu item.
`next`	Keyword	Ends execution of current iteration of a `repeat` loop, beginning next iteration.
`next`	Object modifier	Used with `card` or `background` to refer to the one following the current one.
`newBackground`	System message	Sent to the current card as soon as a background has been created.
`newButton`	System message	Sent to a button as soon as it has been created.
`newCard`	System message	Sent to a card as soon as it has been created.
`newField`	System message	Sent to a field as soon as it has been created.
`newStack`	System message	Sent to the current card as soon as a stack has been created.
`nine`	Constant	String representation of the numerical value 9.
`ninth`	Ordinal	Designates object or chunk number nine within its enclosing set.
`number`	Function	Returns the number of buttons or fields on the current card or background, the number of marked cards, the number of HyperCard menus, the number of menu items in a menu, the number of windows in HyperCard, or the number of a specified type of chunk within a value.
`number`	Property	Determines the number of a background, card, field, or button.
`numberFormat`	Property	Determines the precision with which results of mathematical operations are displayed.
`numToChar`	Function	Returns the character whose ASCII equivalent value is that of the integer passed to it.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`offset`	Function	Returns the number of characters from the beginning of the source string.
`one`	Constant	String representation of the numerical value 1.
`open`	Command	Launches the specified application.
`openBackground`	System message	Sent to a card when you go to it and its background is different from the one you were formerly on.
`openCard`	System message	Sent to a card when you go to it.
`openField`	System message	Sent to an unlocked field when you place the insertion point in it for text editing.
`open file`	Command	Opens the specified file for a `read` or `write` command operation.
`open printing`	Command	Begins a print job.
`open report printing`	Command	Begins a print job for a specified report.
`openStack`	System message	Sent to a card when you go to it and it's in a stack different from the one containing the card you were formerly on.
`optionKey`	Function	Returns the state of the Option key: `up` or `down`.
`oval`	Tool	Name of tool from Tools palette; used with the `choose` command or returned by the `tool` function.
`palette`	Command	Invokes the specified palette XCMD.
`param`	Function	Returns a parameter value from the parameter list passed to the currently executing handler.
`paramCount`	Function	Returns the number of parameters passed to the currently executing handler.
`params`	Function	Returns the entire parameter list passed to the currently executing handler.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`pass`	Keyword	Ends execution of a message handler or function handler and sends the invoking message or function call to the next object in the hierarchy.
`pattern`	Property	Determines the Paint pattern.
`pencil`	Tool	Name of tool from Tools palette; used with the `choose` command or returned by the `tool` function.
`pi`	Constant	The mathematical value pi to 20 decimal places, equal to the number 3.14159265358979323846.
`picture`	Command	Displays the specified picture file in an external window.
`play`	Command	Starts the HyperCard sound-playing feature.
`poly[gon]`	Tool	Name of tool from Tools palette; used with the `choose` command or returned by the `tool` function.
`polySides`	Property	Determines the number of sides created by the Regular Polygon tool.
`pop card`	Command	Returns you to last card saved with the `push card` command.
`powerKeys`	Property	Provides keyboard shortcuts of commonly used painting actions.
`prev[ious]`	Object modifier	Used with `card` or `background` to refer to the one preceding the current one.
`print`	Command	Prints the specified file.
`print card`	Command	Prints the current card or a specified number of cards beginning with the current card.
`printMargin`	Property	Determines or sets the current page print margin.
`printTextAlign`	Property	Determines or sets the text alignment for fields when printing.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`printTextFont`	Property	Determines or sets the text font for fields when printing.
`printTextHeight`	Property	Determines or sets the line height for fields when printing.
`printTextSize`	Property	Determines or sets the text size for fields when printing.
`printTextStyle`	Property	Determines or sets the text style for fields when printing.
`properties`	Property	Returns a list of the names of the palette properties supported by a palette XCMD.
`push`	Command	Saves the identification of a specified card in a LIFO memory stack for later retrieval.
`put`	Command	Copies the value of an expression into a container.
`quit`	System message	Sent to the current card when you choose Quit HyperCard from the File menu (or press Command-Q), just before HyperCard goes away.
`quote`	Constant	The straight double quotation mark character.
`random`	Function	Returns a random integer between 1 and the integer derived from a specified expression.
`read`	Command	Reads a file previously opened with the `open file` command into the local variable `It`. See also `write`.
`rect`	Property	Determines the rectangle property for a variable watcher window and for windows created with the `picture` command.
`rect[angle]`	Property	Determines the rectangle occupied by a specified window, field, or button.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`rect[angle]`	Tool	Name of tool from Tools palette; used with the `choose` command or returned by the `tool` function.
`reg[ular] poly[gon]`	Tool	Name of tool from Tools palette; used with the `choose` command or returned by the `tool` function.
`repeat`	Keyword	Introduces a `repeat` loop, an iterative structure containing a block of one or more statements executed multiple times.
`reportTemplates`	Property	A read-only stack property that returns the names of the report templates for a stack.
`reset menuBar`	Command	Reinstates the default values of all the HyperCard menus and removes any user-defined menus.
`reset paint`	Command	Reinstates the default values of all the painting properties.
`reset printing`	Command	Reinstates the default values of all the printing properties.
`result`	Function	Returns the status of commands previously executed in the current handler. See the `result` function.
`resume`	System message	Sent to the current card when HyperCard resumes running after having been suspended.
`resumeStack`	System message	Sent to the current card when HyperCard returns to a stack.
`return`	Keyword	Returns a value from a function handler or message handler.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`returnInField`	System message	Sent to a field when the Return key is pressed and there is an insertion point or selection in the field.
`returnKey`	Command	Sends any statement in the Message box to the current card.
`returnKey`	System message	Sent to current card when the Return key is pressed.
`right`	Adjective	Specifies right-justified alignment of text in a field.
`right`	Property	Determines or changes the value of item 3 of the `rectangle` property when applied to the specified object or window.
`round`	Function	Returns the number derived from an expression, rounded off to the nearest integer.
`round rect[angle]`	Tool	Name of tool from Tools palette; used with `choose` command or returned by the `tool` function.
`save stack`	Command	Saves a copy of the specified stack with a specified name.
`screenRect`	Function	Returns the size of the screen the HyperCard menu bar is in.
`script`	Property	Retrieves or replaces the script of the specified stack, background, card, field, or button.
`scriptEditor`	Property	Determines the script editor to use.
`scriptTextFont`	Property	Determines the font displayed in the script editor.
`scriptTextSize`	Property	Determines the size of text displayed in the script editor.
`scriptTextStyle`	Property	Determines the style of text displayed in the script editor.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`scroll` (fields)	Property	Determines the amount of material that is hidden above the top of the specified scrolling field's rectangle.
`scroll` (windows)	Property	Determines the position of the window over the card or picture.
`second`	Ordinal	Designates object or chunk number two within its enclosing set.
`seconds`	Function	Returns the number of seconds between midnight, January 1, 1904, and the current time.
`select`	Tool	Name of tool from Tools palette; used with `choose` command or returned by the `tool` function.
`selectedChunk`	Function	Returns a chunk expression describing the selected text in a field.
`selectedField`	Function	Returns an expression describing the field the selected text is in.
`selectedLine`	Function	Returns an expression describing the line in a field where the selected text is.
`selectedText`	Function	Returns the selected text in a field.
`selection`	Container	Currently selected area of text in a field.
`send`	Keyword	Sends a specified message directly to a specified object.
`set`	Command	Changes the state of a specified global, painting, window, or object property.
`seven`	Constant	String representation of the numerical value 7.
`seventh`	Ordinal	Designates object or chunk number seven within its enclosing set.
`sharedHilite`	Property	Determines or sets whether a background button shares the same highlight state on each card.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`sharedText`	Property	Determines if a background field shares the same text on each card. If set to `true` it also sets the `dontSearch` property of the field to `true`.
`shiftKey`	Function	Returns the state of the Shift key: `up` or `down`.
`short`	Adjective	Modifies value returned by `date` function and by `name` and `ID` properties.
`show`	Command	Displays a specified window or object.
`show groups`	Command	Displays a gray line under all text with a `textStyle` property of `group`.
`show cards`	Command	Displays a specified number of cards in the current stack.
`showLines`	Property	Determines whether or not the text baselines are visible in a field.
`show menuBar`	Command	Displays the menu bar if it was hidden.
`showName`	Property	Determines whether or not the name of a specified button is displayed in its rectangle on the screen.
`showPict`	Property	Determines whether or not a specified card or background picture is displayed.
`show titleBar`	Command	Shows the title bar of the current card window if it was hidden.
`sin`	Function	Returns the sine of the angle that is passed to it.
`six`	Constant	String representation of the numerical value 6.
`sixth`	Ordinal	Designates object or chunk number six within its enclosing set.
`size`	Property	Returns the size of a specified stack.
`sort`	Command	Puts all of the cards in a specified stack in a specified order.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`sound`	Function	Returns the name of the sound that is currently playing.
`space`	Constant	The space character (ASCII 32); same as the literal `" "`.
`spray [can]`	Tool	Name of tool from Tools palette; used with the `choose` command or returned by the `tool` function.
`sqrt`	Function	Returns the square root of a number.
`stack`	Object	Generic name of stack object; used with specific name (`go to stack "help"`).
`stacksInUse`	Property	Determines the current list of stacks in the message-passing hierarchy
`startUp`	System message	Sent to the current card (first card of the Home stack) when HyperCard first begins running.
`startUsing`	Command	Specifies a stack to add to the message-passing hierarchy.
`stopUsing`	Command	Specifies a stack to remove from the message-passing hierarchy.
`style`	Property	Determines the style of a specified field or button.
`subtract`	Command	Subtracts the value of an expression from the value in a container.
`suspend`	System message	Sent to the current card when HyperCard is suspended by launching another application with the `open` command.
`suspendStack`	System message	Sent to the current card when you leave an open stack to go to another.
`tab`	Constant	The horizontal tab character (ASCII 9).
`tabKey`	Command	Places the insertion point in the next unlocked field on the current background or card.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`tabKey`	System message	Sent to the current card or a field when Tab key is pressed.
`tan`	Function	Returns the tangent of an angle.
`target`	Function	Indicates the object that initially received the message that initiated execution of the current handler.
`ten`	Constant	String representation of the numerical value 10.
`tenth`	Ordinal	Designates object or chunk number ten within its enclosing set.
`text`	Tool	Name of tool from Tools palette; used with the `choose` command or returned by the `tool` function.
`textAlign`	Property	Determines the alignment of characters created with the Paint Text tool, or those in a field, or those in the name of a button.
`textArrows`	Property	Determines the functions of the arrow keys.
`textFont`	Property	Determines the font of characters created with the Paint Text tool, or those in a field, or those in the name of a button.
`textHeight`	Property	Determines the space between the baseline and characters created with the Paint Text tool or those in a field.
`textSize`	Property	Determines the size of Paint text, or text in a field, or text in the name of a button.
`textStyle`	Property	Determines the style of Paint text, text in a field, text in the name of a button, or text of a menu item.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`the`	Special	Precedes a function name to indicate a function call to one of the built-in functions of HyperCard. You can't call a user-defined function with `the`. Also allowed, but not required, to precede special container names (`the Message box`) and properties.
`then`	Keyword	Follows the conditional expression in an `if` structure to introduce the action clause.
`third`	Ordinal	Designates object or chunk number three within its enclosing set.
`this`	Modifier	Used with `card`, `background`, or `stack` to refer to the current one.
`three`	Constant	String representation of the numerical value 3.
`ticks`	Function	Determines the number of ticks since the Macintosh was turned on or restarted.
`time`	Function	Returns the current time as a text string.
`to`	Preposition	Used to specify ranges (`3 to 5`), connect a message to its destination when used with `send`, specify a format for the `convert` command, assign a container for the `add` command, and connect values to object properties.
`tool`	Function	Returns the name of the current tool.
`top`	Property	Determines or changes the value of item 2 of the `rectangle` property when applied to the specified object or window.
`topLeft`	Property	Determines or changes items 1 and 2 of the value of the `rectangle` property when applied to the specified object or window.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`traceDelay`	Property	Determines or changes the delay between the execution of lines of HyperTalk during a debugger trace.
`true`	Constant	Boolean value resulting from evaluation of a comparative expression and returned from some functions.
`trunc`	Function	Determines the integer part of a number.
`two`	Constant	String representation of the numerical value 2.
`type`	Command	Inserts the specified text at the insertion point.
`unlock screen`	Command	Allows updating of the screen.
`unmark`	Command	Unmarks the specified marked card.
`up`	Constant	Value returned by various functions to describe the state of a key or the mouse button.
`userLevel`	Property	Determines the user level from 1 to 5.
`userModify`	Property	Determines or changes whether or not the user can type into fields or use Paint tools on a stack that has been write-protected.
`value`	Function	Evaluates an expression.
`variableWatcher`	Property	Determines the variable watcher to use.
`vBarLoc`	Property	Determines the location of the vertical bar in the Variable Watcher window.
`version`	Function	Returns the version number of the currently running HyperCard application.
`visible`	Property	Determines whether or not a window, field, or button appears on the screen.
`visual`	Command	Sets up a specified visual transition to the next card opened.
`wait`	Command	Causes HyperCard to pause before executing the rest of the current handler.

[continued]

■ **Table I-1** HyperTalk vocabulary [continued]

Term	Category	Meaning
`wideMargins`	Property	Determines whether or not additional space is displayed in the margins of a specified field.
`width`	Property	Determines or changes the horizontal distance in pixels occupied by the rectangle of the specified button or field.
`windows`	Function	Returns a list of the windows currently available to HyperCard.
`within`	Operator	Tests whether or not a point lies inside a specified rectangle.
`word`	Chunk	Piece of text delimited by spaces in any container or expression.
`words`	Chunk type	Specifies words as type of chunk to the `number` function.
`write`	Command	Copies specified text into a specified disk file starting at a specified point.
`zero`	Constant	String representation of the numerical value 0.
`zoom`	Property	Zooms a window created with the `picture` command in or out.

Glossary

actual parameters: See parameters.

algorithm: A step-by-step procedure for solving a problem or accomplishing a task. Writing HyperTalk handlers or programs in other languages often begins with figuring out a suitable algorithm for a task.

background: A type of HyperCard object; a template shared by a number of cards. Each card with the same background has the same picture, fields, and buttons in its background layer. Like other HyperCard objects, every background has a script. You can place handlers in a background script that you want to be accessible to all the cards with that background.

background button: A button that is common to all cards sharing a background. Compare with **card button.**

background field: A field that is common to all cards sharing a background; its size, position, and default text format remain constant on all cards associated with that background, but its text can change from card to card. Compare with **card field.**

background picture: The graphics in the background layer; the entire picture that is common to all cards sharing a background. You see the background picture by choosing Background from the Edit menu. Compare with **card picture.**

Browse tool: The tool you use to click buttons and to set the insertion point in fields.

button: A type of HyperCard object; a rectangular "hot spot" on a card or background that responds when you click it according to the instructions in its script. For example, clicking a right arrow button with the Browse tool can take you to the next card. See also **background button, card button.**

Button tool: The tool you use to create, change, and select buttons.

card: A type of HyperCard object; a rectangular area that can hold buttons, fields, and graphics. All cards in a stack are the same size. Each layer can contain its own buttons, fields, and graphics.

card button: A button that belongs to a card; it appears on, and its actions apply to, a single card. Compare with **background button.**

card field: A field that belongs to a card; its size, position, text attributes, and contents are limited to the card on which the field is created. Compare with **background field.**

card picture: A picture that belongs to and which applies only to a specific card. Compare with **background picture.**

chunk: A piece of a character string represented as a chunk expression. Chunks can be specified as any combination of characters, words, items, or lines in a container or other source of value.

chunk expression: A HyperTalk description of a unique chunk of the contents of any container or other source of value.

command: A response to a particular message; a command is a built-in message handler residing in HyperCard. See also **external command.**

Command key: The key at the lower-left side of the keyboard that has a propeller-shaped symbol. This key also has an Apple symbol and is sometimes called the *Apple key.*

comments: Descriptive lines of text in a script or program that are intended not as instructions for the computer, but as explanations for people to read. Comments are set off from instructions by symbols called delimiters, which vary from language to language. In HyperTalk, two hyphens `(--)` indicate the beginning of a comment.

constant: A named value that never changes. For example, the constant `empty` stands for the null string, a value that can also be represented by the literal expression `""`. Compare with **variable.**

container: A place where you can store a value (text or a number). Examples are fields, the Message box, the selection, and variables.

control structure: A block of HyperTalk statements defined with keywords that enable a script to control the order or conditions under which specific statements execute.

current: (adj) Applies to the card, background, or stack you're using now. For example, the current card is the one you can see on your screen.

debug: To locate and correct an error or the cause of a problem or malfunction in a computer program, such as a HyperTalk script.

delimiter: A character or characters used to mark the beginning or end of a sequence of characters; that is, to define limits. For example, in HyperTalk double quotation marks act as delimiters for literals, and comments are set off with two hyphens at the beginning of the comment and a return character at the end.

descriptor: The combination of an object's generic name, immediately followed by its particular name, number, or ID number.

dynamic path: A series of extra objects inserted into the path through which a message passes when its static path does not include the current card. The dynamic path comprises the current card, current background, and current stack. Compare **static path.**

expression: A description of how to get a value; a source of value or complex expression built from sources of value and operators.

external command (XCMD): A command written in a computer language other than HyperTalk but made available to HyperCard to extend its built-in command set. External commands can be attached to a specific stack or to HyperCard itself. See also **external function**.

external function (XFCN): A function written in a computer language other than HyperTalk but made available to HyperCard to extend its built-in function set. External functions can be attached to a specific stack or to HyperCard itself. See also **external command**.

factor: A single element of value in an expression. See also **value.**

field: A type of HyperCard object; a container in which you type field text (as opposed to Paint text). HyperCard has two kinds of fields—card fields and background fields.

Field tool: The tool you use to create, change, and select fields.

formal parameters: See **parameter variables.**

function: A named value that HyperCard calculates each time it is used. The way in which the value is calculated is defined internally for the HyperTalk built-in functions, and you can define your own functions with function handlers.

function call: The use of a function name in a HyperTalk statement or in the Message box, invoking either a function handler or a built-in function.

function handler: A handler that executes in response to a function call matching its name.

global properties: The properties that determine aspects of the overall HyperCard environment. For example, `userLevel` is a global property that determines the current user level setting.

global variable: A variable that is valid for all handlers in which it is declared. You declare a global variable by preceding its name with the keyword `global`. Compare with **local variable**.

handler: A block of HyperTalk statements in the script of an object that executes in response to a message or a function call. The first line in a handler must begin with the word `on`, and the last line must end with the word `end`. Both `on` and `end` must be followed by the name of the message or function. HyperTalk has message handlers and function handlers.

hierarchy: See **message-passing hierarchy.**

Home cards: The first five cards in the standard Home stack, designed to hold buttons that take you to stacks, applications, and documents. Choose Home from the Go menu (or press Command-H) to get to the card in the standard Home stack that you've seen most recently. You can also type `go home` in the Message box or include it as a statement in a handler.

HyperTalk: The HyperCard built-in script language for HyperCard users.

identifier: A character string of any length, beginning with an alphabetic character; it can contain any alphanumeric character and the underscore character. Identifiers are used for variable and handler names.

keyboard equivalent key: A key you press together with the Command key to issue a menu command.

keyword: Any one of the 14 words that have a predefined meaning in HyperTalk. Examples of keywords are `on`, `if`, `do`, and `repeat`.

layer: The order of a button or field relative to other buttons or fields on the same card or background. The object created most recently is ordinarily the topmost object (on the front layer).

literal: A string of characters intended to be taken literally. In HyperTalk, you use quotation marks (" ") as delimiters to set off a string of characters as a literal, such as the name of an object or a group of words you want to be treated as a text string.

local variable: A variable that is valid only within the handler in which it is used. (Local variables need not be declared.) Compare with **global variable.**

loop: A section of a handler that is repeated until a limit or condition is met, such as in a repeat structure.

message: A string of characters sent to an object from a script or the Message box, or that HyperCard sends in response to an event. Messages that come from the system—from events such as mouse clicks, keyboard actions, or menu commands—are called system messages. Examples of HyperTalk messages are `mouseUp`, `go`, and `push card`. See also **handler.**

Message box: A container that you use to send messages to objects or to evaluate expressions.

message handler: A handler that executes in response to a message matching its name.

message-passing hierarchy: The ordering of HyperCard objects that determines the path through which messages pass.

metasymbol: A word used in a syntax statement as a placeholder for an element that is different for each specific use of the statement. For example, the metasymbol *filename* is used to show where you put the name of a file you want a command to act on. In this book, metasymbols are shown in italics.

number: A character string consisting of any combination of the numerals 0 through 9, optionally including one period (.) representing a decimal value. A number can be preceded by a hyphen or a minus sign to represent a negative value.

object: An element of the HyperCard environment that has a script associated with it and that can send and receive messages. There are five kinds of HyperCard objects: buttons, fields, cards, backgrounds, and stacks.

object descriptor: Designation used to refer to an object. An object descriptor is formed by combining the name of the type of object with a specific name, number, or ID number. For example, `background button 3` is an object descriptor. Stacks do not have a number or ID number, so only the name can be used for a stack descriptor.

object properties: The properties that determine how HyperCard objects look and act. For example, the `location` property of a button determines where it appears on the screen.

online help: Assistance you can get from an application program while it's running. In HyperCard, online help refers to the HyperCard disk-based Help system.

operator: A character or group of characters that causes a particular calculation or comparison to occur. In HyperTalk, operators operate on values. For example, the plus sign (+) is an arithmetic operator that adds numerical values.

Paint text: Text you type using the Paint Text tool. Paint text can appear anywhere, while regular text must appear in a field created with the Field tool. Paint text is part of a card or background picture.

Paint tool: Any HyperCard tool you use to make pictures. Paint tools include Lasso, Brush, Spray, Eraser, and many others.

painting properties: The properties that control aspects of the HyperCard painting environment, which is invoked when you choose a Paint tool. For example, the `brush` property determines the shape of the Brush tool.

palette: A small window that displays icons or patterns you can select by clicking. You can see two of the HyperCard palettes, the Tools palette and the Patterns palette, simply by "tearing off" their respective menus. To see the navigator palette, type `palette "navigator"` in the Message box. See also tear-off menu.

parameters: Values passed to a handler by a message or function call. Any expressions after the first word in a message are evaluated to yield the parameters; the parameters to a function call are enclosed in parentheses or, if there is only one, it can follow `of`.

parameter variables: Local variables in a handler that receive the values of parameters passed with the message or function call initiating the handler's execution.

picture: Any graphic or part of a graphic, created with a Paint tool or imported from an external file, that is part of a card or background.

pixel: Short for *picture element*; the smallest dot you can draw on the screen. The position of the pointer is often represented by two numbers separated by commas. These numbers are horizontal and vertical distances of the pointer from the left and upper edges of the card window, measured in pixels. The upper-left corner of the screen has the coordinates `0,0`.

point: (1) A location on the screen described by two integers, separated by a comma, representing horizontal and vertical offsets, measured in pixels from the upper-left corner of the card window or (in the case of the card window itself) of the screen. (2) In printing, the unit of measurement of the height of a text character; one point is about 1/72 of an inch. When you select a font, you can also select a point size, such as 10 point, 12 point, and so on.

power key: One of a number of keys on the Macintosh keyboard you can press to initiate a menu action when a Paint tool is active. Power keys are enabled when you choose Power Keys from the Options menu or you check Power Keys on the Preferences card in the Home stack.

properties: The defining characteristics of any HyperCard object and of the HyperCard environment. For example, setting the user level to Scripting changes the `userLevel` property of HyperCard to the value 5. Properties are often selected as options in dialog boxes or on palettes, or they can be set from handlers.

Recent: A special dialog box that holds pictorial representations of the last 42 unique cards viewed. Choose Recent from the Go menu to get the dialog box. Also, an adjective describing the card you were on immediately prior to the current card, as in `recent card`.

recursion: The repetition of an operation or group of operations. Recursion occurs when a handler calls itself.

resource fork: The part of a file that contains resources such as fonts, icons, sounds, and so on.

regular text: Text you type in a field. You use the Browse tool to set an insertion point in a field and then type. Regular text is editable and searchable, while Paint text is not.

script: A collection of handlers written in HyperTalk and associated with a particular object. You use the script editor to add to and revise an object's script. Every object has a script, even though some scripts are empty, that is, they contain nothing.

script editor: A window in which you can type and edit a script. The title bar of the script editor describes the object to which the script belongs. You can use the Edit menu, the Script menu, and keyboard commands to edit text in the script editor. See also handler, object, and script.

search path: When you open a file from within HyperCard, HyperCard attempts to locate the stack, document, or application you want by searching the folders listed on the appropriate Search Paths card in the Home stack. Each line on a Search Paths card indicates the location of a folder, including the disk name (and folder and subfolder names, if any). This information is called a *search path.* Items in a search path are separated by a colon, like this:

```
my disk:HyperCard folder:
my stacks:
```

Search Path cards: Three cards in the Home stack used to store information about the location of stacks, documents, and applications that you open while HyperCard is running. See also **search path**.

selection: A container that holds the currently selected area of text. Note that text found by the `find` command is not selected.

shared text: Field text that appears on every card in a background. Shared text can be edited only from the background layer. Text in shared fields cannot be searched.

source of value: The most basic HyperTalk expressions; the language elements from which values can be derived: constants, containers, functions, literals, and properties.

stack: A type of HyperCard object that consists of a collection of cards; a HyperCard document.

statement: A line of HyperTalk code inside a handler. A handler can contain many statements. Statements within handlers are first sent as messages to the object containing the handler and then to succeeding objects in the message-passing hierarchy.

static path: The message-passing route defined by an object's own hierarchy. For example, the static path followed by a message sent to (but not handled by) a button would include the card to which the button belongs, the background associated with that card, and the stack containing them. Compare **dynamic path.**

string: A sequence of characters. You can compare and combine strings in different ways by using operators. In HyperTalk, for example, `23 + 23` will result in `46`; but `23 & 23` will result in `2323`.

syntax: A description of the way in which language elements fit together to form meaningful phrases. A syntax statement for a command shows the command in its most generalized form, including placeholders (sometimes called metasymbols) for elements you must fill in as well as optional elements.

System file: Software your computer uses to perform its basic operations.

system message: A message sent by HyperCard to an object in response to an event such as a mouse click, keyboard action, or menu command. Examples of HyperCard system messages are `mouseUp`, `doMenu`, and `newCard`.

target: The object that first receives a message.

tear-off menu: A menu that you can convert to a palette by dragging the pointer beyond the menu's edge. HyperCard has two tear-off menus—Tools and Patterns. When torn off these menus are referred to as palettes.

text field: See **field.**

text property: A quality or attribute of a character's appearance. Text properties include style, font, and size.

tick: Approximately one-sixtieth (1/60) of a second. The `wait` command assumes a value in ticks unless you specify seconds by adding `secs` or `seconds`.

tool: An implement you use to do work. HyperCard has tools for browsing through cards and stacks, creating text fields, editing text, making buttons, and creating and editing pictures.

user level: A property of HyperCard, ranging from 1 to 5, that determines which HyperCard capabilities are available. You can select the user level on the Preferences card in the Home stack. Each user level makes all the options from the lower levels available, and also gives you additional capabilities. The five user levels are Browsing, Typing, Painting, Authoring, and Scripting.

value: A piece of information on which HyperCard operates. All HyperCard values can be treated as strings of characters—they are not formally separated into types. For example, a numeral could be interpreted as a number or as text, depending on what you do with it in a HyperTalk handler.

variable: A named container that can hold a value consisting of a character string of any length. You can create a variable to hold some value (either numbers or text) simply by using its name with the put command and putting the value into it. HyperCard has local variables and global variables. Compare with **constant**.

window properties: The properties that determine how the Message box and the Tools and Patterns palettes are displayed. For example, the `visible` property determines whether or not the specified window is displayed on the screen.

Index

& (concatenate) 110, 112
&& (concatenate with space) 110
( ) (grouping) 108
* (multiply) 108
+ (plus) 108
- (minus) 108
-- (double = hyphen) comment character 19, 107
/ (divide) 108
< (less than sign) 109
<= (less than or equal to) 109
> (greater than) 109
^ (exponent) operator 108
<> (not equal) 109
>= (greater than or equal to) 109
≠ (not equal sign) 109
≤ (less than or equal to sign) 109
≥, >= (greater than or equal to sign) 109

A

abbreviated 80
abs function 260
accessing XCMDs and XFCNs 443
actual parameters 73
add command 158
after 230
all 243
ambiguous stack descriptors 84
and 108
annuity *See also* compound 260
answer 159
    It as destination 100
answer file 159
answer for ask command 165
any 82
arithmetic operators 107, 112
arrowKey
    command 162
    system message 129, 162
ASCII codes 503-506
ask 164
    It as destination 100
ask file 164
ask password command 164
assigning menu names 87
atan function 261
autoHilite property 334
autoTab property 336

B

backgrounds
    current 18
    defined 18
    descriptors for 78-82
    properties 318
background buttons
    sharedHilite 410
background fields
    sharedText 411
barn door close visual effect 252
barn door open visual effect 252
beep command 166
before 230
blindTyping property 337
bottom property 338
bottomRight property 339
brush property 340
brush tool 167
bucket tool 167
built-in function 95

button Info dialog box 29
button properties 322-324 *See also specific button property*
button text
text alignment 418
`button` tool 167
`buttonCount` palette property 217
buttons
defined 16
descriptors for 78-82
editing scripts of 28-38
messages to 54-55
system messages and 122-124

C

`Can't understand` error message 52
`cantAbort` property 341, 344
`cantDelete` property 342
`cantModify` property 343
card fields 78
Card Info dialog box 78
card picture 407
scroll property 407
scrolling 407
card properties 319. *See also specific card property*
`card window`
current 90
`defined` 21
`rectangle` property 397, 401
`scroll` property 407
card window properties
scroll 407
cards
current 18
defined 18
descriptors for 78-82
editing scripts of 28-38
system messages and 126-132
`centered` property 345
`char[acter]` (chunk) 114
`checkerboard` visual effect 252
`checkMark` property 346, 376
checkpoints 39, 41, 180
`choose` command 167
chunk
as a destination 119
defined 112
nonexixtant 119
chunk expression 112, 119
ranges in 116
syntax of 113
click
location 169
`click` command 169
`clickChunk` function 263, 264
`clickLoc` function 265
`clickText` function 266
`close file` command 171, 212
`close printing` command 172, 213
`close` system message 127
`close window` command 173
`closeBackground` system message 131
`closeCard` system message 49, 127
`closeField` system message 125
`closePalette` system message 127
`closePicture` system message 127
`closeStack` system message 131
closing external windows 481
`cmdChar` property 347
`commandKey` function 267
`commandKey` system message 129
commands 155. *See also* messages *or specific commands*
defined 155
overriding 136
syntax notation 157
syntax summary 516
`commands` palette property 217
comment character (--) 19
`comment` command 35
commenting scripts 35
complex expressions 106-107
`compound` function 268. *See also* `annuity`
concatenate(&) 110

concatenate with space (&&) 110
constants 493-494
  defined 94
containers
  chunk expressions and 117-119
  defined 98
  fields 98
  Message box 102-103
  the selection 100
  variables 73-75, 99
contains operator 109
control structures 135-154
controlKey 174
  message parameter values 175
controlKey system message 130
convert command 176
cos function 269
create *See also*, disable; disabled; enable; enabled; text; textStyle; markChar; cmdChar; menuMsg; and put
create menu command 178
create stack command 179
  in a new window 179
  with background 179
creating menus 178
current hierarchy 56, 57
current objects 18
cursor property 348
curve tool 167
cutCard system message 133

D

date function 270
debug checkpoint command 180
debugger 39-45
  checkpoints 41
  command summary 45
  defined 39
  exiting 41
  menu 39
  stepping through scripts 40
  tracing through scripts 41
debugger property 349
debugger tools 42
debugger windows 42-44
debugging environment 39
delete
  menu command 181
  menu item 181
deleteBackground system message 131, 133
deleteButton system message 123
deleteCard system message 127, 133
deleteField system message 124
deleteStack system message 131, 133
dial command 183
dialog window layer 484
disable command 184 *See also*, enable.
diskSpace function 271
dissolve visual effect 252
div operator 108
divide command 185
do command 186
do keyword 152
document window layer 484
doMenu
  command 187
  intercepting 156
  intercepting handler 188
  system message 130
dontSearch property 350
dontWrap property 351
double - hyphen comment character (--) 19, 35, 107
down constant 494
drag command 189
dragSpeed property 352
Draw Centered setting 329, 345
Draw Multiple setting 329, 381
dynamic path 63-66
  go command and 63
  invoking 63
  send keyword and 63, 65-66

E

edit script command 190
editBkgnd property 353
eighth (ordinal) 81
else keyword 149-151
empty constant 99, 494
enable command 191 *See also*, disable.
enabled property 354
end keyword 19, 137, 141, 148
enterKey Command 192
enterKey system message 128
environmental properties 326
equal (=) 108
eraser tool 167
example XCMD 487
exit keyword 68, 138, 142, 147
exp function 272
exp1 function 272
exp2 function 273
export paint command 193
expressions 93, 103, 105, 106-119
  complex expressions 106-112
external commands and functions 441-492
  defined 442
external window events 475-480
external windows 462-468, 474
  callbacks 462-468
  closing 481
  event handling 468

F

factors 107
false constant 494
field properties 320-322, 438 *See also specific field property.*
field text 418
  alignment 418
field tool 167
fields 98
  as containers 98
  defined 16
  descriptors for 78, 83
  system messages and 124-126
fifth (ordinal) 81
filled property 355
find 197, 349
  dontSearch 350
first (ordinal) 81
five (constant) 81
fixedLineHeight property 356
floating layer 484
formFeed constant 494
foundChunk *See also* find.
foundChunk function 273
foundField function 274
foundLine function 275
foundText function 275
four (constant) 81
fourth (ordinal) 81
FreeSize property 357
function call 20, 258
function handlers 26, 140-143
  example 143
  keywords in 140-143
  overriding 140
  parameter passing into 73
  user-defined 140
function keyword 141
functionKey command 198
functionKey system message 129
functions 141, 257-313 *See also specific function.*
  defined 257
  length 276
  redefining 258
  syntax notation 259
  syntax summary 520, 523
  value 312
  windows 313

G

get command 199

It as destination 100
global keyword 152
global properties 326-328, 431 *See also specific global property*
language 366
global statement 152
global variables 43, 99
go command 200-201
greater than or equal to (≥, >=) 109
greater than (>) 109
grid property 358
Grid setting 329

H

handlers 57, 122
as subroutines 67
calling handlers 67-69
defined 19
function handlers 20-26, 74, 140, 143
intercepting commands 75, 156
intercepting messages 71
message handlers 20
nesting 68
recursion 68-69
sharing 69-70
statements within 19
hBarLoc property 359
height property 360
help command 201
help system message 132
hide system message 132
hideIdle property 361
hideUnused property 362
hiding windows 202
hiding objects 202-204
hierarchy, message-passing 52-66, 69-72
current 52
defined 52
objects in 54-55
hilite property 363 *See also sharedHilite and autoHilite*
hilitedButton palette property 217
HyperCard
enhancements since HyperCard 1.2.5 1-14
performance hints 495, 501

I

icon property 364
ID property 81, 365
forms of 81
identifying a stack 83-84
full pathname 83
idle 49
idle system message 128
if 148-151
import paint command 205
in 86, 113, 157
Info menu 28
intercepting messages 71-73, 378
doMenu 378
menu messages 378
interrupting executing handlers 68
into (preposition) 230
iris close visual effect 252
iris open visual effect 252
is in operator 109
is not in operator 110
is not operator 109
is operator 108
It 100
items as chunk expressions 114

K

keyDown system message 130
keyword 50, 57, 135, 154 *See also specific keyword*
defined 135
function 140, 141

L

language property 366
lasso tool 167
last (ordinal) 82

layered buttons and fields 54
left property 367
length function 276
less than or equal to (<=) 109
less than (<) 109
line tool 167
lineFeed constant 494
lines as chunk expressions 115
lineSize property
  line tool and 368
  shape tool and 368
literal defined 94
ln function 276
ln1 function 277
loc property 369
local variables 43, 99
location property 169, 369
  point 369
lock screen command 206
lockMessages property 371
lockRecent property 372
lockScreen property 373
lockText property 374
log2 function 277
long (adjective) 80
longWindowTitles property 375

M

mark command 207
markChar 376
marked property 377
max function 278
me 83
menu
  commands 122
  disable 184
  enable 191
  number of 285
menu command 187
menu function 279
menu item names 87, 88
menu item properties 331-332, 383, 426
  See also specific menu-item property
menu items 24, 230
  adding messages for 230
  defined 24
  disable 184
  enable 191
  refering to by number 88
  setting text style of 426
menu messages 24, 230
  defined 24
  intercepting 378
menu names 87, 88
menu numbers 87
menu properties
  checkMark 376
  enabled 354
  name 383
menuMessage 378, 379 *See also* put, doMenu
menus 24, 77
  controlling through HyperTalk 88, 89
  creating menus 88
  defined 24
  disabling 184
  enabling 191
  refering to by number 87
menus function 279
message sending order 18
Message box 50, 102
message handlers 19, 136-139
  example 139
  keywords in 136, 139
  syntax of 136
message name 73
Message Watcher 42-43
Message Watcher properties 332-333 *See also specific properties*
message-passing hierarchy
  current 56-57
  user-defined 59, 62
  using 69-72

messages 48, 50-75
commands 18
from external commands 51
handling 48
intercepting 71
keywords in 50
matching message names 51-52
receiving 51
resulting from commands 51
sending 48
sent to a field 54, 124-126
sent to a locked field 49
sent to buttons 49, 54, 122
sent to current card 126-132
system messages 49, 52
sent to a card 49
messageWatcher property 380
metasymbol definitions 524-530
middle (ordinal) 82
min function 280
minus (-) 108
mod operator 108
modems
dial command and 183
mouse function 281
mouseClick function 282
mouseDown system message 49, 123, 125, 127
mouseDownInPicture system message 127
mouseEnter system message 49, 54, 123, 126
mouseH function 283
mouseLeave system message 124, 126
mouseLoc function 283
mouseStillDown system message 123, 125, 127
mouseUp system message 123, 126, 127
mouseUpInPicture system message 127
mouseV function 284
mouseWithin system message 124, 126
moveWindow system message 132
multiple property 381
multiply (*)108
multiply command 209
multiSpace property 382

N

name property 80, 383
naming menus 87
naming objects 78-79, 84
naming stacks 85
navigator palette 5, 24, 216
newBackground system message 130, 133
newButton system message 123
newCard system message 127, 133
newField system message 124
newStack system message 131, 133
next 82
next repeat statement 147
nine (constant) 81
ninth (ordinal) 81
not equal (<>) 109
not operator 108
number function 285. *See also* number *property*
number handling defined 97
number property 385. *See also* number *function*
numberFormat property 386
numbers 98
defined 96
decimal string precision 97
numberFormat property and 97
SANE numeric values 96
numToChar function 287. *See also* charToNum

O

object ID number 80, 81
object descriptors 78
combining 86
descriptor phasing 79

object hierarchy 53
message passing in 75
object Info dialog box 317
object names 79
assigning names 79
object numbers 81
integers 81
numeric constants 81
ordinal constants 81
reassigning 82
special ordinals 82
tab order 82
object properties 316
ID property 81
name property 79
objects 16, 77
background buttons 78
backgrounds 18
buttons 16
cards 18
fields 16, 77
generic names 78
referring directly to 86
script 19
stacks 18
Objects menu 28
of 86, 113, 157
offset function 288
on 19, 137
one (constant) 81
open
document with application 210
open command 210
open file command 212, 255
open printing command 213
open report printing command 214
openBackground system message 131
openCard 49
openCard system message 127
openField 49
openField system message 125
opening
documents 212
files for reading or writing 212
openPalette system message 127
openPicture system message 127
operators 107
and expression type 112
comparision operators 107
numneric values 112
precedence of 111, 206
optionKey function 289. *See also* commandKey
or operator 108
ordinal constant 113
oval tool 167
overriding commands 136

P

Paint text 99, 418
alignment 418
Paint Text tool 418
painting properties 329-330. *See also specific painting property*
palettes
navigator 5, 24, 216
user-defined 5, 216
palette command 216
palette properties 217
param function 290, 291. *See also* paramCount, params
parameter list 74, 137
parameter passing 73-75
to handlers 75
parameter variables 73-74
parameters
defined 73
in function handlers 74
params function 292
pass keyword 51, 63, 138, 142
pasteCard system message 133
pattern property 387
Patterns palette 387
pencil tool 167
pi constant 494

picture command 218
globalRect 219, 220
loc 220
rect 219
scale 221
scroll 220
plain visual effect 252
play command 223
playing sound 223
plus (+) 108
point 169
polygon tool 167
polySides property 388
pop card command 224. *See also* pushcard
powerKeys property 389
precedence of operators 111
prepositions 263
previous 82
print
document with application 225
expression 225
field 225
marked cards 227
print card command 227
print command 225
printing
fields 225
reports 214
printMargins property 390
printTextAlign property 391
printTextFont property 392
printTextHeight property 393
printTextSize property 394
printTextStyle property 395
properties
defined 95
syntax 337
palette 217
Protect Stack dialog 342, 343
push command 229
put command 119, 230. *See also*: enabled; text; textStyle; cmdChar; *and* menuMsg

Q

quit system message 131
quote constant 494

R

random function 293
ranges of chunks 116
read command 233
file 233
It as destination 233
limits 234
until character 233
reassigning object numbers 82
recent 83
rect property 396
rectangle properties 320, 321, 324, 397
bottom 338
bottomRight 339
height 360
left 367
right 401
top 427
topLeft 428
width 439
rectangle tool 167
recursion 68
redefining commands 156
redefining functions 258
refering to windows 90
refering to fields 98
refering to menu items 88
refering to menus 87
repeat statement 144-148
for 145
forever 145
until 145
while 146
with 146

reportTemplates property 400
reserved words (keywords) 136-154
reset menuBar command 235
reset paint command 236
reset printing command 236
result function 294
resume system message 131, 133
resumeStack system message 131
retrieving properties 316
return constant 494
return keyword 138, 143
return statement 142
returnInField system message 125
returnKey command 237
returnKey system message 128
right property 401
round function 296
round rectangle tool 167

S

save command 237
save stack 237
screen rectangles 324, 398
screenRect function 296
script 19, 402
script comments 19
script debugger 39
script editor 31, 37-38
  automatic formatting 36
  breaking long line statements 37
  command summary 38
  comments 35
  formatting scripts 36
  manipulating text 33
  opening multiple scripts 31
  replacing text 34
  saving a script 29
  script size 37
  searching 33
  shortcuts 30
script property 402
scriptEditor property 403
scripting userLevel 28
scripts
  background buttons 29
  background fields 29
  background, editing the script of 28
  button, editing the script of 29
  closing 29, 30
  current card, editing the script of 28
  fields, editing the script of 29
  function handlers within 20
  getting to object scripts 28
  opening 28-29
  saving 29
  scripting shortcuts 29
  shortcuts for opening 30
  stack, editing the script of 28
scriptTextFont property 404
scriptTextSize property 405
scriptTextStyle property 406
scriptWindowRects 14
scroll down visual effect 252
scroll left visual effect 252
scroll property
  card windows 407
  fields 408
scroll right visual effect 252
scroll up visual effect 252
searching
  find command 194
second (ordinal) 81
seconds function 297. *See also* convert
select command 238
select tool 167
selected text 101
selectedChunk function 298
selectedField function 299
selectedLine function 300
selectedLoc function 301
selectedText function 302
selection 100
send keyword 69, 153
set command 239. *See also properties*
setting properties 239, 316
seven (constant) 81

seventh (ordinal) 81
sharedHilite property 410
sharedText property 411
sharing handlers 69
shiftKey function 303
short (adjective) 80
shortcuts
  closing scripts 30
  opening scripts 29
show command. See also hide and picture commands241
  marked cards 243
  background picture 240
  card picture 240
  groups 240
show system message 132
showing windows 240
showLines property 412
showName property 413
showPict property 414
shrink from bottom visual effect 252
shrink from center visual effect 252
shrink from top visual effect 252
sin function 303
six (constant) 81
sixth (ordinal) 81
size property 415
soft return 37
sort command 244
sound function 304. *See also* play
sources of value
  constants 94
  functions 95
  literals 94
  numbers 96
  properties 95
space constant 494
special object descriptors 82-83
special ordinals 82, 113
spray can tool 167, 555
sqrt function 305
stack
  current 18
  defined 18
  descriptors for 83, 84
  identifying 83
  naming 85
  refering to 84
stack name 83
stack properties 317-318. *See also specific stack property*
stacks function 305
stacksInUse property 416
start using command 246
startUp system message 127, 133
statements 19
  as messages 50
  defined 19
  formatting 36
static path 63
stepping 40
stop using command 247
stretch from bottom visual effect 252
stretch from center visual effect 252
stretch from top visual effect 252
style property 417
subroutine calls 68
  calling handler 67
  subroutine handler 67
subtract command 247
suspend system message 131
suspendStack system message 131
syntax
  notation for commands 156
  notation for functions
  summary for commands 516-520
  summary for functions 520-523
system messages 49, 121-132
  entry point in hierarchy 55
  order of 133

T

tab constant 494
tab order 82

tabKey command 248. *See also* tabKey *system message*
tabKey system message 126, 128
tan function 306
target function 58, 307
temporary checkpoints 41
ten (constant) 81
tenth (ordinal) 81
text operators 112
text tool 167
textAlign property 418
textArrows property 419
textFont property 420
textHeight property 421
textSize property 422
textStyle property 424, 426
the result 138
there is a operator 110
there is not a operator 110
third (ordinal) 81
three (constant) 81
ticks function 308
time function 309
to 116
tool function 310. *See also* choose
top property 427
traceDelay property 429
tracing 41
TRUE constant 494
trunc function 311
two (constant) 81
type command 249

U

unlock screen command 250
  with visual effect 250
unmark command 251
up constant 494
user-defined (custom) menus 87
user-defined message-passing hierarchy 59-62
  adding stacks 246
  deleting stacks 247
  handlers in 61
userLevel property 430
userModify property 431

V

value function 312
values 93-103
variable name 99
Variable Watcher 43-44
Variable Watcher properties 332-333. *See also specific properties*
variables 99-100
  and numbers 96
  as containers 99
  defined 99
  global 99
  It 100
  local 99
  parameter variables 73-74, 100
  values stored in 96
variableWatcher property 432
vBarLoc property 433
venetian blinds visual effect 252
version property 434
visible property 436-438
visual command 252
visual effects 252
vocabulary list 531

W

wait command 254
wideMargins property 438
width property 439
window layers defined 484
window properties 330-331. *See also specific properties*
windows 21, 77
  defined 21
windows function 313
wipe down visual effect 252
wipe left visual effect 252
wipe right visual effect 252

wipe up visual effect 252
within operator 110
words as chunk expressions 114
write command 255. *See also* open file; close file, *and* read

X

XCmdBlock 448
XCMDs and XFCNs 442
  accessing 443
  attaching the resource 447
  callback fields 450
  callback procedures and functions 451-473
  debugger callbacks 473
  entryPoint 450
  external window callbacks 462-468
  field callbacks 457
  guidlines for writing 446
  HyperTalk callbacks 452
  in the message-passing hierarchy 444
  inArgs 451
  invoking 443
  maximum parameters 444
  memory callbacks 454
  outArgs 451
  paramCount field 449
  parameter block 448
  params array 449
  passFlag 450
  passing back results 449
  passing information 448
  passing parameters 449
  request 450
  result 451
  returnValue 449
  script editor callbacks 470-471
  special XCmdBlock values 481-482
  string callbacks 454-455
  string conversion callbacks 455-457
  text editing callbacks 464, 469
  uses for 443
  utility callbacks 459-461
  Variable Watcher callbacks 472
  XCmdBlock 448
XTalkObject fields
  bkgndID 483
  buttonID 484
  cardID 483
  fieldID 484
  objectKind 483
  stackNum 483
XTalkObject structure defined 483-484

Z

zero..ten constant 494
zoom close visual effect 252
zoom in visual effect 252
zoom open visual effect 252
zoom out visual effect 252

Writer: Steve Schwander

Illustration: Mary Prusmack
Mil Madamba

Thanks to Barbara Smyth.

Thanks to Kevin Calhoun, Mike Holm,
Steve Maller, Donald Olson, and
Jeremy Bornstein for their technical assistance.

A very special thanks to Alan Spragens for creating the first Macintosh *HyperCard Script Language Guide*, which served as an important part of the source material for this manual.

U90-614-001A